Security & Exploitation

Operating Systems Fall 2017

RPISEC - 12/11/17
RPISSEC

The Computer Security Club @ RPI

- [https://rpis.ec ; irc.rpis.ec](https://rpis.ec ; irc.rpis.ec)

We Teach All Things Security

- Weekly Friday Meetings + INTROSEC Tuesday Meetings

We compete in CTFs

- “Hacking Competitions”
We Play CTFs
We (sometimes) Win CTFs
We Attend Conferences (with style)
Security?

A *complete* understanding of how a system functions so we can *subvert* or *fix* any flaws or weaknesses it might have
Security?

- Binary Exploitation
- Reverse Engineering
- Cryptography
- Web & Networking
- Penetration Testing
- Lockpicking
- Malware Analysis / Dev.
- Forensics
- Privacy
Security?

- **Binary Exploitation → Today’s Focus**
- Reverse Engineering
- Cryptography
- Web & Networking
- Penetration Testing
- Lockpicking
- Malware Analysis / Dev.
- Forensics
- Privacy
- …
Binary Exploitation

- **Pwning** [ˈpʌɪŋ] v.
  - Using bugs and flaws in an executable to gain control over a process and eventually, the system it runs on.

- Requires knowledge of the compiler, operating system, and architecture.
  - Need to know what makes the machine tick.
Agenda

- C Programming Language
- Stack Overflows & Return Oriented Programming
- Bug Classes
- Heap Allocators (if time permits)
C Programming Language

- “C is quirky, flawed, and an enormous success” — Dennis Ritchie
- Keeps the security industry in business
  — (Well, and the mess that is the Modern Web)
Why is C such a Culprit?

- **C is a very low-level language**
  - Compiles directly to machine code
    - Code is very fast
  - Code has full control over memory
    - Does exactly what you want tell it to (usually)
Why is C such a Culprit?

• But as they say...
  – With great power comes dangling pointers
• Very easy to make dangerous mistakes
  – segmentation fault (core dumped)

push   edi
call   sub_314623
test   eax, eax
jz     short loc_31306D
cmp    [ebp+arg_0], ebx
jnz    short loc_313066
mov    eax, [ebp+var_70]
cmp    eax, [ebp+var_84]
jb     short loc_31306B
sub    eax, [ebp+var_84]
push   esi
push   eax
push   edi
mov    [ebp+arg_0], eax
call   sub_31486A
test   eax, eax
jz     short loc_31306D
push   esi
push   eax
mov    [ebp+arg_0], eax
mov    esi, 1D0h
push   esi
push   eax
mov    [ebp+arg_4], eax
call   sub_314623
test   eax, eax
jz     short loc_31306D
cmp    [ebp+arg_0], esi
jz     short loc_31308F
loc_313066: ; CODE XREF: sub_312FD0 ; sub_312FD8+51
push    0Dh
call    sub_31411B
loc_31306D: ; CODE XREF: sub_312FD0 ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg     short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
loc_31307D: ; CODE XREF: sub_312FD0
and    eax, UserRt
or     eax, 80000000h
loc_31308C: ; CODE XREF: sub_312FD0
mov    [ebp+arg_4], eax
Let's Start Breaking Stuff

- Demo server:

  ```
  ssh opsyst@warzone.rpis.ec
  password: opsyst
  
  opsys@warzone:~$ labs/lab1
  Welcome to our secure service.
  Enter your name:
  Bob
  Bob, you have $37.
  opsys@warzone:~$
  ```

- Try out the Super Secure Bank™
What's in a Name?

• Let's send a lot of As

opsys@warzone:~$ labs/lab1
Welcome to our secure service.
Enter your name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA,
you have $1094795585.
Segmentation fault
opsys@warzone:~$

• Looks like the bank wasn't as secure as they thought... Let's look at the source!
The Gift that Keeps on Getting

- What could have caused that?
  - It is the evil `gets` ( ... ) function
  - Keeps reading input, never stops (Buffer overflow)
  - DO NOT EVER USE

- Similar functions
  - `strcpy` ( ... )
  - `strcat` ( ... )
  - `scanf("%s", ...)`
Quick Review from Comp Org

- Program executes using a stack
  - Each function call creates a stack frame
    - Contains local variables
    - Pointer to previous frame
    - Return address

<table>
<thead>
<tr>
<th>Lower Memory</th>
<th>0x53</th>
<th>0x6f</th>
<th>0x6e</th>
<th>0x6e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x79</td>
<td>0x00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Higher Memory</th>
<th>0x25</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf8</td>
<td>0xf6</td>
<td>0xff</td>
<td>0xbf</td>
<td></td>
</tr>
<tr>
<td>0x39</td>
<td>0x85</td>
<td>0x04</td>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Start of char name [32]
- Contains name

- End of name
- int money = 37
- Padding
- Ptr to previous stack frame
- Return address
- Other frames
Corrupting the Stack

- As we give more input, we fill up the stack frame
Corrupting the Stack

- As we give more input, we fill up the stack frame
- Fill it up too much and it starts corrupting things
  - Buffer overflow
Corrupting the Stack

• Our money is now 0x41414141
  – $1094795585
• But what if we keep going...

```
0x41 0x41 0x41 0x41
0x41 0x41 0x41 0x41
0x41 0x41 0x41 0x41
...```

```
0x41 0x41 0x41 0x41
0x41 0x41 0x41 0x41
0x41 0x41 0x41 0x41
0x41 0x41 0x41 0x00
```

```
- - - -
0xf8 0xf6 0xff 0xbf
0x39 0x85 0x04 0x08
```

```
- - - -
- - -
- - -
```

- Start of char name[32]
- End of name
- int money = 37
- Ptry to previous stack frame
- Return address
Corrupting the Stack (Segfault Edition)

- Start of char name[32]
- End of name
- int money = 37
- Ptr to previous stack frame
- Return address
- Other frames

- If we go too far we corrupt the return address
  - Program crashes because it is incorrect
  - We will talk more about this later

- Lower Memory
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - ...
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - 0x41 0x41 0x41 0x41
  - Higher Memory
    - 0x00 - - -
Breaking the Bank

• Let's try and set the money to $3735928559
• Quick note on endianness
  – Most x86 machines are little endian
  – Meaning the order of bytes for numbers is reversed in memory
  – 0x01020304 would be

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>0x03</td>
<td>0x02</td>
<td>0x01</td>
</tr>
</tbody>
</table>

• Steps to success
  – Convert 3735928559 to hex
  – Split bytes apart and put them in the correct endianness
• Wait... How do we write bytes like that?
Breaking the Bank

• How to send this kind of input
  – Python can write raw bytes using the **hex escape**
  – You can use python to craft the payload and pipe it

```
python -c 'print "A"*32+\\x01\\x02\\x03\\x04" | .\lab1
```

• Steps to success (again)
  – Convert **3735928559** to hex
  – Split the bytes apart and put them in the right endianness
  – Use python to pipe the bytes into the bank
Breaking the Bank

- (python -c 'print ...'; cat)| ./bank
- Keeps stdin open
- Solution:

```python
(python -c "\nprint "A"*32+"\xef\xbe\xad\xde";cat) | ./level1
```
Execution Basics

- CPU’s execute machine code, typically 1 instruction at a time.
#include <stdio.h>

int main(int argc, char** argv) {
    unsigned int x, y;
    x = 1;
    y = 2;

    x = y + 1;
    y = x + 1;

    return 0;
}
Execution Basics

**EIP:** Instruction Pointer

**ESP:** Stack Pointer

**EBP:** Base Pointer

**General Purpose:**
- EAX
- EBX
- ECX
- EDX

**“String” Registers:**
- ESI
- EDI
Execution Basics

EIP: 0x80483ed
ESP: 0xbffff6fc
EBP: 0x0

General Purpose:
EAX: 0x1
EBX: 0xb7fcd000
ECX: 0xd48a11f8
EDX: 0xbffff724

“String” Registers:
ESI: 0x0
EDI: 0x0
Execution Basics

EIP: 0x80483ee
ESP: 0xbffffff6f8
EBP: 0x0

General Purpose:
EAX: 0x1
EBX: 0xb7fcd000
ECX: 0xd48a11f8
EDX: 0xbffffff724

“String” Registers:
ESI: 0x0
EDI: 0x0

```assembly
[0x80483ed]
main:
(fcn) sym.main 45
; var int local_0_1  @ ebp-0x1
; var int local_1  @ ebp-0x4
; var int local_2  @ ebp-0x8
; DATA XREF from 0x08048307 (sym.main)
push ebp
mov ebp, esp
sub esp, 0x10
mov dword [ebp-local_2], 1
mov dword [ebp-local_1], 2
mov eax, dword [ebp-local_1]
add eax, 1
mov dword [ebp-local_2], eax
mov eax, dword [ebp-local_2]
add eax, 1
mov dword [ebp-local_1], eax
mov eax, 0
leave
ret
```
Execution Basics

EIP: 0x80483f0
ESP: 0xbffffff6f8
EBP: 0xbffffff6f8

General Purpose:
EAX: 0x1
EBX: 0xb7fcd000
ECX: 0xd48a11f8
EDX: 0xbffffff724

“String” Registers:
ESI: 0x0
EDI: 0x0
Execution Basics

EIP: 0x80483f3
ESP: 0xbfffff6e8
EBP: 0xbfffff6f8

General Purpose:
EAX: 0x1
EBX: 0xb7fcd000
ECX: 0xd48a11f8
EDX: 0xbffff724

“String” Registers:
ESI: 0x0
EDI: 0x0
Execution Basics

EIP: 0x80483fa
ESP: 0xbfffff6e8
EBP: 0xbfffff6f8

Memory View @ EBP:

0xbfffff6f8: 00 00 00 00
0xbfffff6f4: ?? ?? ?? ??
0xbfffff6f0: 01 00 00 00
Execution Basics

EIP: 0x8048401
ESP: 0xbfffff6e8
EBP: 0xbfffff6f8

Memory View @ EBP:

0xbfffff6f8: 00 00 00 00 00
0xbfffff6f4: 02 00 00 00 00
0xbfffff6f0: 01 00 00 00 00
Execution Basics

EIP: 0x8048401
ESP: 0xbfffff6e8
EBP: 0xbfffff6f8

Memory View @ EBP:

0xbfffff6f8: 00 00 00 00
0xbfffff6f4: 02 00 00 00
0xbfffff6f0: 01 00 00 00

And so on...
#include <stdio.h>

int main(int argc, char** argv) {
    unsigned int x, y;
    x = 1;
    y = 2;
    x = y + 1;
    y = x + 1;
    printf("%u", x);
    return 0;
}
Control Flow Basics

What happens here?
Control Flow Basics

What happens here?

1: **Push** address of next instruction onto the stack. This is the “return address”

2: **Jump** to call-target address
Control Flow Basics

- **Libraries (libc)**
- **ELF Executable**
  - `.text segment`
  - `.data segment`
- **Heap**
- **Stack**

**EIP:** @printf

**ESP**

- `<Address of: mov eax, 0>`
- `<Old Stack Frame #1>`
- `<Old Stack Frame #2>`
- `<...>`
- `<Bottom of Memory>`

```
push ebp
mov ebp, esp
and esp, 0xffffffff0
sub esp, 0x20
mov dword [esp + 0x18], 1
mov dword [esp + 0x1c], 2
mov eax, dword [esp + 0x1c]
add eax, 1
mov dword [esp + 0x18], eax
mov eax, dword [esp + 0x18]
add eax, 1
mov dword [esp + 0x1c], eax
mov eax, dword [esp + 0x18]
mov dword [esp + 4], eax
mov dword [esp], 0x8048500
call sym.imp.printf; [a]
leave
ret
```
Control Flow Basics

What happens here?

Pop address on top of the stack into EIP
Control Flow Basics

What happens here?

Pop address on top of the stack into EIP

ESP:  < Return Address >  
<  Old Values #1  >

EIP:  0x08048466
Control Flow Basics

What happens here?

**Pop address on top of the stack into EIP**

**ESP:**  
< Old Values #1  >  
< Old Values #2  >  

**EIP:**  
< Return Address >
Returning Back to the Bank

• Let's try and access the bank's secret vault

• How can we turn the crash into something useful?
  – Use what you now know about the return address
Returning Back to the Bank

- **Without corruption:**
  - At the end of the function, it **returns**
  - **0x08048539** is popped off the stack and stored in **eip**
  - Control goes to that address

- **We want to change this**

### Memory Layout

<table>
<thead>
<tr>
<th>Lower Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x41</td>
</tr>
<tr>
<td>0x41</td>
</tr>
<tr>
<td>0x41</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>Higher Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x39</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>
Returning Back to the Bank

- **Corrupted like before:**
  - Last time it crashed, can you see why?
  - Function *returns* like before
  - **0x41414141** is popped off the stack and stored in **eip**
  - Control goes there
  - But where is that???

- **Segmentation Fault**
Getting Into the Vault

• By changing the **return address**, we can take control of the program.
• Need to get the address of the **secretVault()** function.
• Overwrite the return address with **secretVault()**.
• **Return** from the function.
• Profit (Shell on their system)
Getting Into the Vault

- **Our goal:**
  - Set return address to 0x80484f1
  - Use correct endianness
- Use the same technique as before to overwrite the stack data

<table>
<thead>
<tr>
<th>Lower Memory</th>
<th>0x41</th>
<th>0x41</th>
<th>0x41</th>
<th>0x41</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
<td>0x41</td>
</tr>
<tr>
<td></td>
<td>0xf1</td>
<td>0x84</td>
<td>0x04</td>
<td>0x08</td>
</tr>
<tr>
<td>Higher Memory</td>
<td>0x00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Bug Classes

- Buffer Overflow
- Integer Overflow
- Printf (format string) bugs (you heard me!)
- Time of Check vs Time of Use
int createAccounts(int records) {
    Account *buffer = malloc(records * sizeof(Account));
    for (int i = 0; i < records; i++) {
        buffer[i] = malloc(sizeof(Account));
    }
    accounts = buffer;
}

Spot the bug
int createAccounts(int records) {
    Account *buffer = malloc(records * sizeof(Account));
    for (int i = 0; i < records; i++) {
        buffer[i] = malloc(sizeof(Account));
    }
    accounts = buffer;
}
Integer Overflow

• Can induce undefined behavior
• Sometimes hard to exploit
void saveImage(char *imagename, uint8_t *buffer, uint len)
{
    printf("Saving image named: ");
    printf(imagename);
    FILE *fp = fopen(imagename, "wb+");
    write(fp, buffer, len);
    printf("\nDone saving image.\n");
}

Spot the Bug

RPSEC - 12/11/17  Operating Systems Fall 2017
Format String Bugs

```c
void saveImage(char *imagename, uint8_t *buffer, uint len)
{
    printf("Saving image named:\n");
    printf(imagename);
    FILE *fp = fopen(imagename, "wb+");
    write(fp, buffer, len);
    printf("\nDone saving image.\n");
}
```
Format String Bugs

- The string can contain arbitrary format specifiers \%x, \%n, \%d, \%f, \%llu, which will read random stack contents
- ... So what?
Format String Bugs

• `%n` specifier has the ability to write the number of characters printed to a variable.

• If no variable is present, it’ll pick whatevers on the stack, where the variable would be.

• Can now write memory arbitrarily.
Format String Bug
void loadFirmwareFromFile(char *filename) {
    if (checkSHA1(filename, "7af0aeffa...38bf54ae3")) {
        FILE *fp = fopen(filename, "rb");
        mapImage(fp);
    }
}

push edi
call sub_314623
test eax, eax
jz loc_31306D
cmp [ebp+arg_0], ebx
jnz loc_313066
mov eax, [ebp+var_70]
cmp eax, [ebp+var_84]
jb loc_313066
sub eax, [ebp+var_84]
push esi

push esi
push eax
push edi
push [ebp+arg_0]
call sub_31486A
test eax, eax
push esi
lea eax, [ebp+arg_0]
push eax
mov esi, 1D0h
esi push [ebp+arg_4]
push edi
call sub_314623
test eax, eax
jz loc_31306D
cmp [ebp+arg_0], esi
ja loc_31306F

loc_31306E:
push 0Dh
call sub_31411B

loc_31306D:
call sub_3140F3

loc_31307D:
call sub_3140F3

loc_31308C:

Time-of-check vs Time-of-Use

- (Or more generally, race conditions)
- Caused by lack of proper locking over a shared resource
- Can’t be really dangerous, was used for iPhone jailbreaks historically
That’s all!

Questions?

Get in Touch:
irc.rpis.ec
contact@rpis.ec

loc_31306E:
 push 0DH
 call sub_31411B

loc_31306D:
call sub_3140F3
 test eax, eax
 jmp short loc_31307D
 call sub_3140F3

loc_31307D:
call sub_3140F3
 add eax, [ebp+arg_1]
 or eax, 80000000h

loc_31308C:
mov [ebp+arg_4], eax