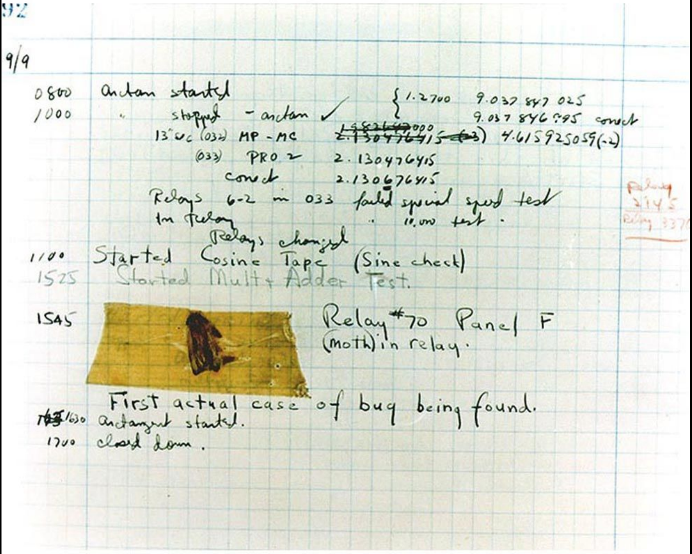
OSY.SSI[2015][9]

Software Lab: Code Injection

Question

Photo # NH 96566-KN (Color) First Computer "Bug", 1947



Remember: Bugs are our friends

Lesson: Bugs tell us something interesting about *how things work*.

Pedagogy: We're gonna cause bugs.

Programme

- Review of lhe compilation process, disassemble, analyse
- Stack overflow
- Control flow hijacking and exploits
- Break
- Bypassing protections
- Advanced injections / Attack planning

Beware: Vote.

Democracy yay

You must choose today what exam you'd like best.

./0 Short group presentation on a research paper

./1 Super Hacker Smash Brawl

`doodle.com/poll/7c4335pgczumagid`

I'll do my best to make it happen. Voting will close at midnight.

Important remark

Never analyse malware running directly on a production machine.

Important remark

Never analyse malware running directly on a production machine.

Important remark

Never analyse malware running directly on a production machine.

The examples are tailored for 32 bit GNU/Linux, on which security safeguards are disabled.

Important remark

Never analyse malware running directly on a production machine.

The examples are tailored for 32 bit GNU/Linux, on which security safeguards are disabled.

- (As `root`) `echo 0 > /proc/sys/kernel/randomize_va_space`

Important remark

Never analyse malware running directly on a production machine.

The examples are tailored for 32 bit GNU/Linux, on which security safeguards are disabled.

- (As `root`) `echo 0 > /proc/sys/kernel/randomize_va_space`
- `ulimit -s unlimited`

Important remark

Never analyse malware running directly on a production machine.

The examples are tailored for 32 bit GNU/Linux, on which security safeguards are disabled.

- (As `root`) `echo 0 > /proc/sys/kernel/randomize_va_space`
- `ulimit -s unlimited`
- Compile with `-fno-stack-protector -z execstack -ggdb -O0`

Important remark

Never analyse malware running directly on a production machine.

The examples are tailored for 32 bit GNU/Linux, on which security safeguards are disabled.

- (As `root`) `echo 0 > /proc/sys/kernel/randomize_va_space`
- `ulimit -s unlimited`
- Compile with `-fno-stack-protector -z execstack -ggdb -O0`

Cheating? We will re-enable them later on ;)

Table of Contents

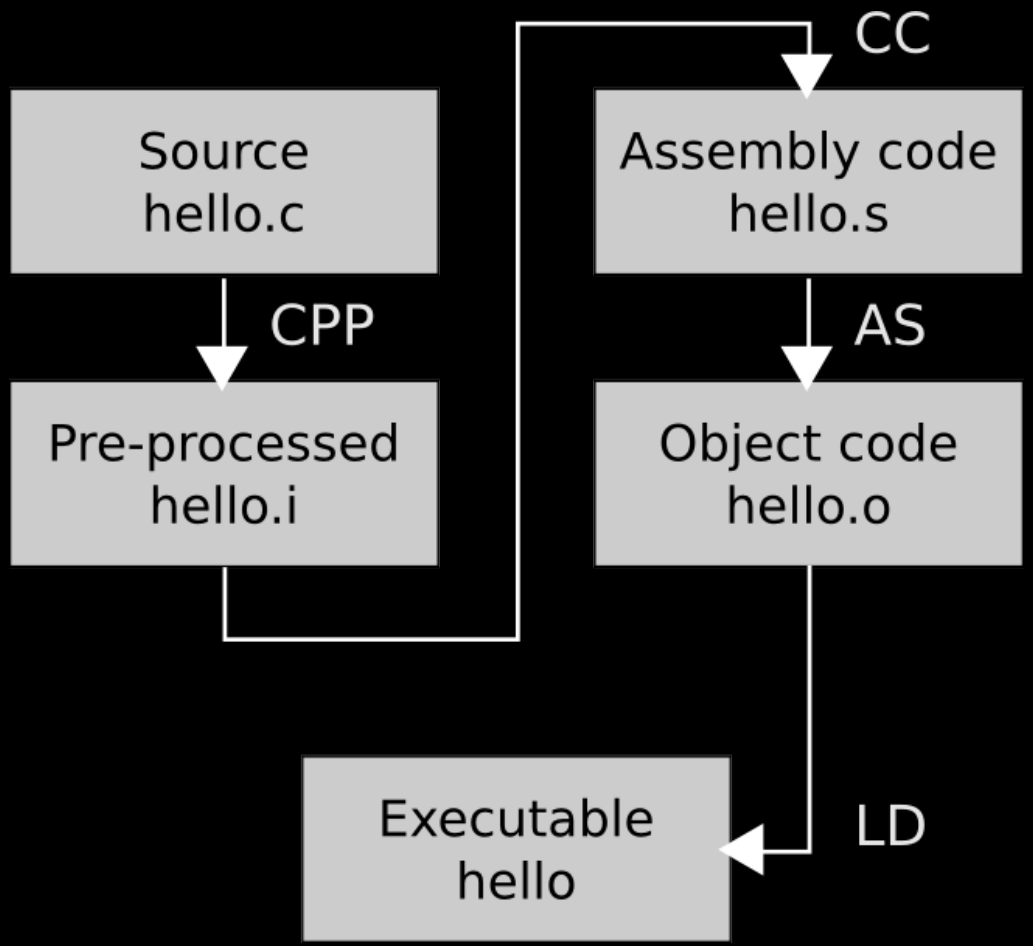
How do we make programs?

- Pay an intern to write code
- ???
- Profit!

How do we make programs when we're poor?

- Write code yourself
- Pre-processing, compiling, assembling, linking

The C compilation process



A simple C program

Task:

A simple C program

Task:

- Write a program printing “Hello World!” in C

A simple C program

Task:

- Write a program printing “Hello World!” in C
- Use `puts` instead of `printf`

A simple C program

Task:

- Write a program printing “Hello World!” in C
- Use `puts` instead of `printf`
- Compile with `gcc`: `gcc -o <outputfile> <sourcefile.c>`

A simple C program

Task:

- Write a program printing “Hello World!” in C
- Use `puts` instead of `printf`
- Compile with `gcc`: `gcc -o <outputfile> <sourcefile.c>`
- Analyse with `objdump -d`

A simple C program

Task:

- Write a program printing “Hello World!” in C
- Use `puts` instead of `printf`
- Compile with `gcc`: `gcc -o <outputfile> <sourcefile.c>`
- Analyse with `objdump -d`
- Compare to `gcc -S`

A simple C program

Task:

- Write a program printing “Hello World!” in C
- Use `puts` instead of `printf`
- Compile with `gcc`: `gcc -o <outputfile> <sourcefile.c>`
- Analyse with `objdump -d`
- Compare to `gcc -S`

Question:

A simple C program

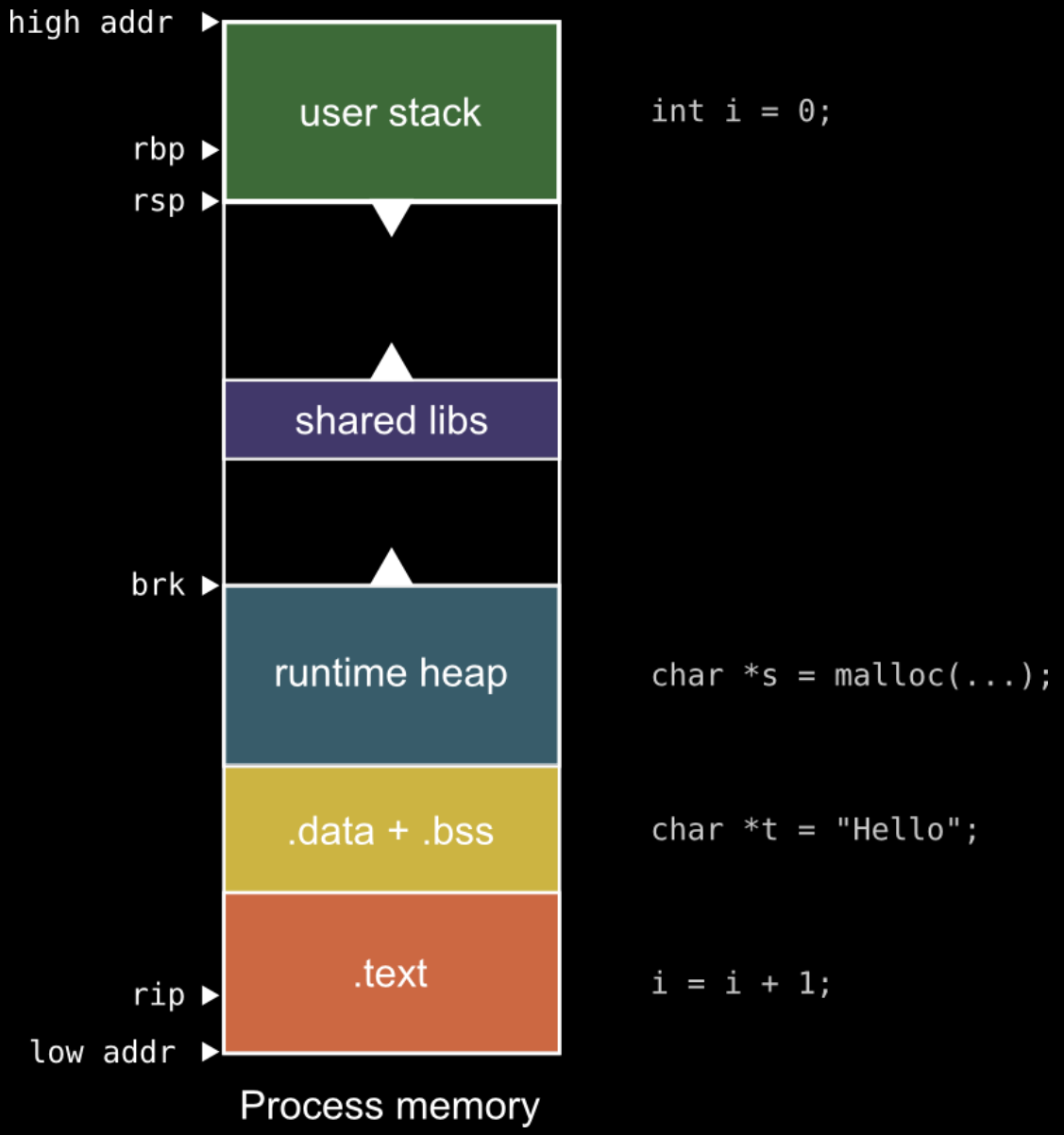
Task:

- Write a program printing “Hello World!” in C
- Use `puts` instead of `printf`
- Compile with `gcc`: `gcc -o <outputfile> <sourcefile.c>`
- Analyse with `objdump -d`
- Compare to `gcc -S`

Question: What is the “exit value”?

Recall: The stack

The program's *Weltanschauung*



A less simple C program

Task:

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use `gdb` to trace)

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use `gdb` to trace)

Q1:

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use `gdb` to trace)

Q1: How are arguments sent to the function?

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use gdb to trace)

Q1: How are arguments sent to the function?
Q2:

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use gdb to trace)

Q1: How are arguments sent to the function?
Q2: How is the result returned?

A less simple C program

Task:

- Write in C a program printing “Hello World!” ten times and analyse it
- Write a function printing “Hello World!” n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use gdb to trace)

Q1: How are arguments sent to the function?
Q2: How is the result returned?
Q3:

A less simple C program

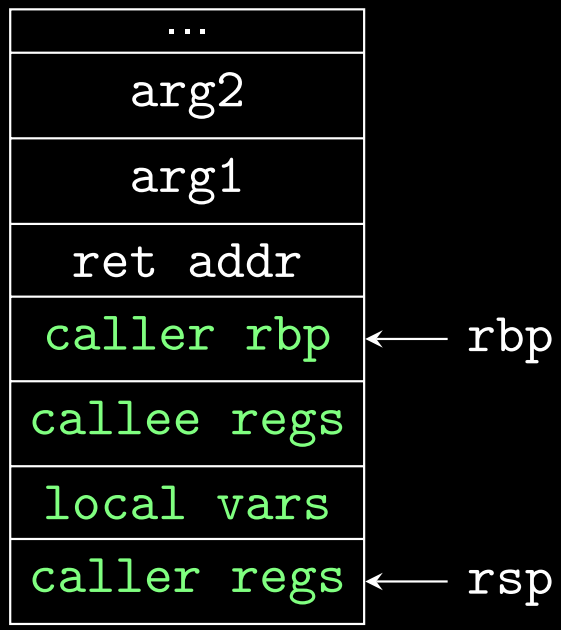
Task:

- Write in C a program printing "Hello World!" ten times and analyse it
- Write a function printing "Hello World!" n times, call it with $n = 10$
- Write a function taking n, printing it (use `printf`), and returning n^2
- (Bonus : use gdb to trace)

Q1: How are arguments sent to the function?
Q2: How is the result returned?
Q3: How does the program continue after the function returned?

Recall: How functions work

The stack



Recall: Syscalls

Task:

Recall: Syscalls

Task:

- Write/read an assembly program printing “Hello World!” using syscall 4

Recall: Syscalls

Task:

- Write/read an assembly program printing “Hello World!” using syscall 4
- Compile with `nasm -f elf hello.asm`

Recall: Syscalls

Task:

- Write/read an assembly program printing “Hello World!” using syscall 4
- Compile with `nasm -f elf hello.asm`
- Link with `ld hello.o -o hello`

Recall: Syscalls

Task:

- Write/read an assembly program printing “Hello World!” using syscall 4
- Compile with `nasm -f elf hello.asm`
- Link with `ld hello.o -o hello`
- Analyse with `objdump -D`

Recall: Syscalls

Task:

- Write/read an assembly program printing “Hello World!” using syscall 4
- Compile with `nasm -f elf hello.asm`
- Link with `ld hello.o -o hello`
- Analyse with `objdump -D`

Table of Contents

Disassembly

Recall that assembly is just a representation of the program.
We will use `objdump` to understand binaries.
Such a tool is called a disassembler.

Mystery file

Task:

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:**

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:** use `objdump -D -b binary -mi386 mystery`

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:** use `objdump -D -b binary -mi386 mystery`
- What it is?

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:** use `objdump -D -b binary -mi386 mystery`
- What it is?
- What does it do?

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:** use `objdump -D -b binary -mi386 mystery`
- What it is?
- What does it do?
- Test it with `mystery_run.c`

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:** use `objdump -D -b binary -mi386 mystery`
- What it is?
- What does it do?
- Test it with `mystery_run.c`

This kind of (short) program is known as a *shellcode*.

Mystery file

Task:

- Analyse the `mystery` file whose content is

```
\x31\xc9\xf7\xe1\x51\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\xb0\x0b\xcd\x80
```

- **Hint:** use `objdump -D -b binary -mi386 mystery`
- What it is?
- What does it do?
- Test it with `mystery_run.c`

This kind of (short) program is known as a *shellcode*.

Thousands of shellcodes: `http://www.exploit-db.com/`.

Table of Contents

Our friend the segmentation fault

Task:

Our friend the segmentation fault

Task:

- Read the program `password.c`. How does it work?

Our friend the segmentation fault

Task:

- Read the program `password.c`. How does it work?
- Compile it (even without flags)

Our friend the segmentation fault

Task:

- Read the program `password.c`. How does it work?
- Compile it (even without flags)
- Play with it.

Our friend the segmentation fault

Task:

- Read the program `password.c`. How does it work?
- Compile it (even without flags)
- Play with it.
- Cause a segmentation fault.

Our friend the segmentation fault

Task:

- Read the program `password.c`. How does it work?
- Compile it (even without flags)
- Play with it.
- Cause a segmentation fault.
- What is a segmentation fault?

Our friend the segmentation fault

Task:

- Read the program `password.c`. How does it work?
- Compile it (even without flags)
- Play with it.
- Cause a segmentation fault.
- What is a segmentation fault?

Note: the program is vulnerable to a denial of service attack.

Our friend the segmentation fault

Task:

Our friend the segmentation fault

Task: Gain access without knowing the password.

Our friend the segmentation fault

Task: Gain access without knowing the password.
Note:

Our friend the segmentation fault

Task: Gain access without knowing the password.
Note: the program is vulnerable to priviledge escalation.

Our friend the segmentation fault

Task: Gain access without knowing the password.
Note: the program is vulnerable to priviledge escalation.

BTW, how would you find the password if you needed it?

From segfault to hijack

Task:

From segfault to hijack

Task:

- Run `password` in `gdb`.

From segfault to hijack

Task:

- Run `password` in `gdb`.
- Cause a segmentation fault.

From segfault to hijack

Task:

- Run `password` in `gdb`.
- Cause a segmentation fault.
- Show registers with `info registers`.

From segfault to hijack

Task:

- Run `password` in `gdb`.
- Cause a segmentation fault.
- Show registers with `info registers`.
- What happens? What can we do with it?

From segfault to hijack

Task:

From segfault to hijack

Task:

- Read, compile and run `hijack.c`

From segfault to hijack

Task:

- Read, compile and run `hijack.c`
- Cause a segmentation fault

From segfault to hijack

Task:

- Read, compile and run `hijack.c`
- Cause a segmentation fault
- Run the program in `gdb`

From segfault to hijack

Task:

- Read, compile and run `hijack.c`
- Cause a segmentation fault
- Run the program in `gdb`
- Try to hijack `eip`

From segfault to hijack

Task:

- Read, compile and run `hijack.c`
- Cause a segmentation fault
- Run the program in `gdb`
- Try to hijack `eip`
- What happens?

From segfault to hijack

Task:

- Read, compile and run `hijack.c`
- Cause a segmentation fault
- Run the program in `gdb`
- Try to hijack `eip`
- What happens? Gulliver.

From segfault to hijack

Task:

- Read, compile and run `hijack.c`
- Cause a segmentation fault
- Run the program in `gdb`
- Try to hijack `eip`
- What happens? Gulliver.

This is called *control flow hijacking*.

Table of Contents

From hijack to exploit

Why stop there?

From hijack to exploit

Why stop there?

Instead of padding, why not insert code?

From hijack to exploit

Why stop there?

Instead of padding, why not insert code? A shellcode for instance?

And then what do we do?

Arbitrary code execution (ACE)

Task:

Arbitrary code execution (ACE)

Task:

- Write an exploit for `hijack` that runs a shell.

Arbitrary code execution (ACE)

Task:

- Write an exploit for `hijack` that runs a shell.
- Modify the stack : `export OSYSSI=rocks`

Arbitrary code execution (ACE)

Task:

- Write an exploit for `hijack` that runs a shell.
- Modify the stack : `export OSYSSI=rocks`
- Does it still work?

Arbitrary code execution (ACE)

Task:

- Write an exploit for `hijack` that runs a shell.
- Modify the stack : `export OSYSSI=rocks`
- Does it still work? Why?

Arbitrary code execution (ACE)

Task:

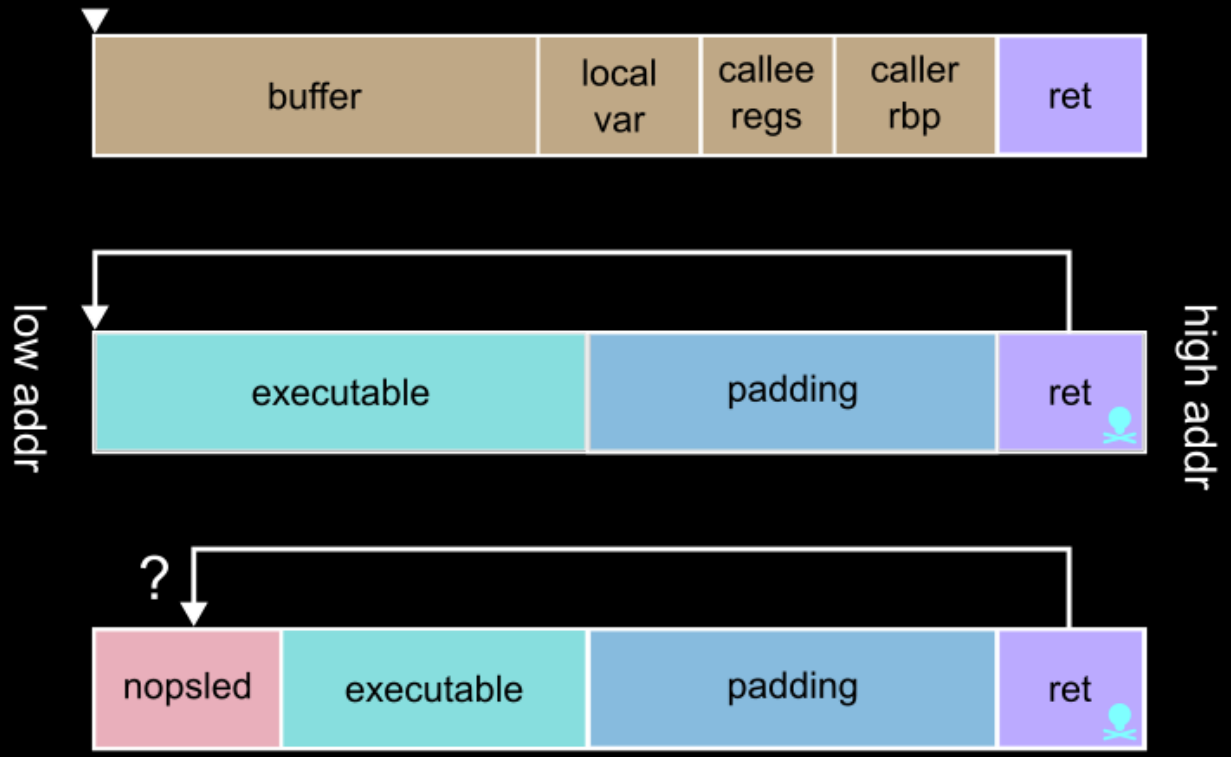
- Write an exploit for `hijack` that runs a shell.
- Modify the stack : `export OSYSSI=rocks`
- Does it still work? Why? Fix it.

Arbitrary code execution (ACE)

Task:

- Write an exploit for `hijack` that runs a shell.
- Modify the stack : `export OSYSSI=rocks`
- Does it still work? Why? Fix it.
- Make it more robust using the *nop sled trick*.

Arbitrary code execution (ACE)



ACE + PE

Task:

ACE + PE

Task:

- Give `hijack` the `root` delegation (SUID)

ACE + PE

Task:

- Give `hijack` the `root` delegation (SUID)
- Why would any program have that?

ACE + PE

Task:

- Give `hijack` the `root` delegation (SUID)
- Why would any program have that?
- Exploit `hijack` to launch a root shell.

ACE + PE

Task:

- Give `hijack` the `root` delegation (SUID)
- Why would any program have that?
- Exploit `hijack` to launch a root shell.

What are the consequences?

Question: what key security principle was forgotten?

Traditional stack smashing

Attackers would usually:

Traditional stack smashing

Attackers would usually:

- Design a *bind* or *connect-back* shell-code (why?)

Traditional stack smashing

Attackers would usually:

- Design a *bind* or *connect-back* shell-code (why?)
- Send the exploit to a vulnerable remote server (why?)

Traditional stack smashing

Attackers would usually:

- Design a *bind* or *connect-back* shell-code (why?)
- Send the exploit to a vulnerable remote server (why?)
- Gain root access on the server.

Traditional stack smashing

Attackers would usually:

- Design a *bind* or *connect-back* shell-code (why?)
- Send the exploit to a vulnerable remote server (why?)
- Gain root access on the server.

Just a bug?

CVE-2015-3876

2 Oct 2015

`libstagefright` buffer overflow in Android up to 5.1.1 allows remote attackers to execute arbitrary code via crafted metadata in a MP3 or MP4 file.

Still unpatched as we speak. Probably never will be for older Android versions.

CVSS score: 9.3/10.
"Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit."

Table of Contents

Different approaches

- Not making vulnerable code in the first place

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material
  - Antivirus, IDS, firewalls,...

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material
  - Antivirus, IDS, firewalls,...
- Refusing non authorised code and stack execution

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material
  - Antivirus, IDS, firewalls,...
- Refusing non authorised code and stack execution
  - Signed code, DEP, NX, W$\oplus$X, ...

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material
  - Antivirus, IDS, firewalls,...
- Refusing non authorised code and stack execution
  - Signed code, DEP, NX, W$\oplus$X, ...
- Shuffling the memory model

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, “memory-safe” languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material
  - Antivirus, IDS, firewalls,...
- Refusing non authorised code and stack execution
  - Signed code, DEP, NX, W$\oplus$X, ...
- Shuffling the memory model
  - ASLR, ASCII-armoured addresses...

Different approaches

- Not making vulnerable code in the first place
  - Testing, static analysis, "memory-safe" languages, coding guidelines,...
- Detecting stack smashing as it happens and blocking it
  - Canaries, segmentation, SEH, SEHOP,...
- Analysing user actions for suspicious material
  - Antivirus, IDS, firewalls,...
- Refusing non authorised code and stack execution
  - Signed code, DEP, NX, W$\oplus$X, ...
- Shuffling the memory model
  - ASLR, ASCII-armoured addresses...

What works? At what cost?

Note: this is just **one** sort of vulnerability.

Break, now.

After the break:

- Bypassing protections
- Advanced techniques? or Attack planning?