Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Software Security (Part 3)

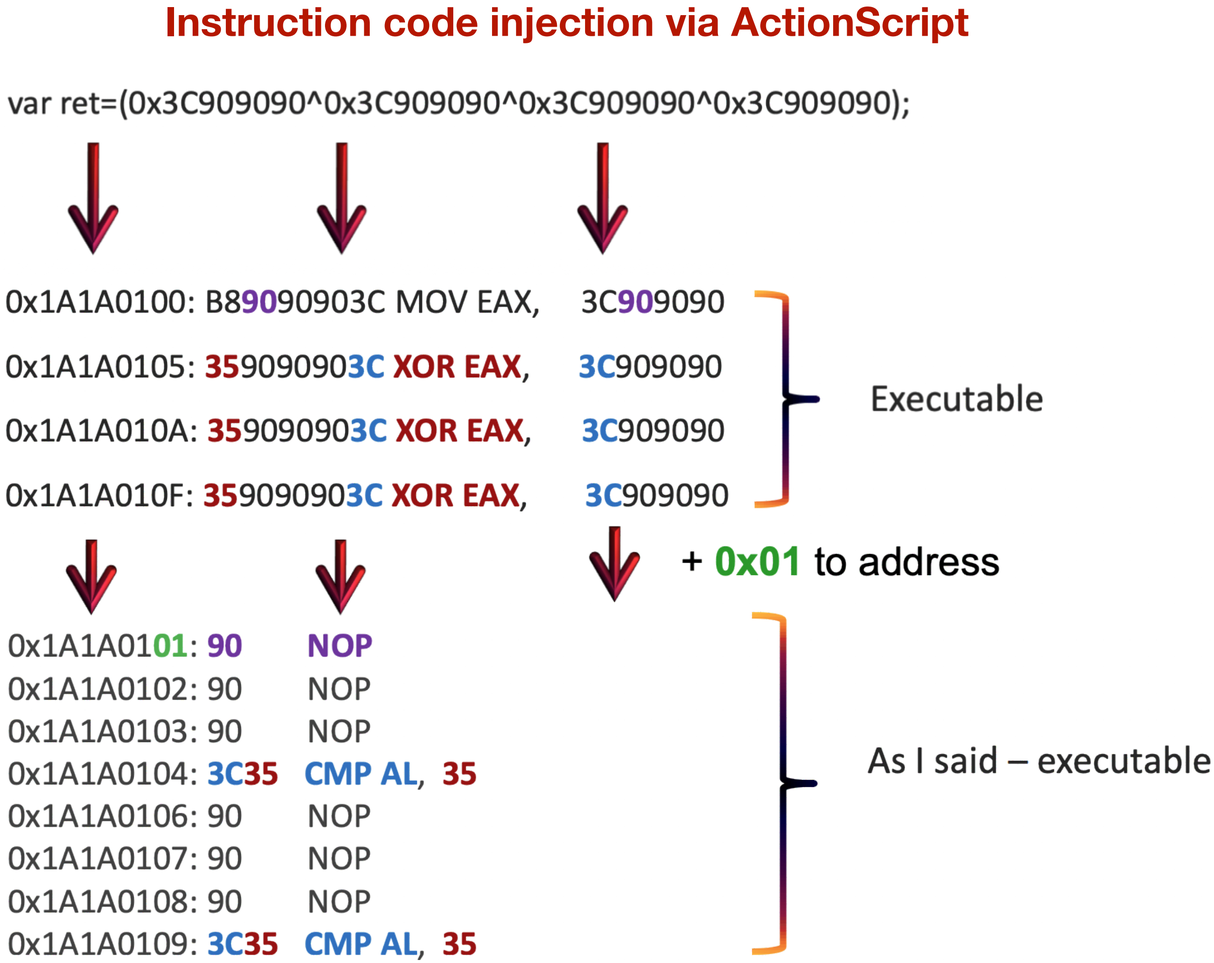
Hugo Pacheco
hpacheco@fc.up.pt

Stealing control is harder

- An attacker needs to inject malicious code (shellcode) in memory & change control to jump to that area and execute the malicous code
- Modern Oses implement countermeasures such as: (more on this later)
  - Data Execution Prevention (DEP) or W ^ X (Write or Execute)
    - An executable memory page cannot be written
    - A writable memory page cannot be executed
    - Some exceptions for “Just-In-Time” Compilation ⇒ 
  - Address Space Layout Randomisation (ASLR)
    - The memory locations of critical parts of a program are “shuffled” in each execution

JIT Spraying

- “Just-In-Time” compilation common for bytecode languages, e.g., Java, JavaScript, Adobe Flash
- **Goal:** more frequently used code is compiled to machine code “on-the-fly”
- **Attack:** abusing the JIT compiler to insert malicious code at runtime:
  - compiling constants (less suspicious) + **jumping to “half addresses”!**
  - executable code ⇒ DEP ☠
  - shellcode spray ⇒ ASLR ☠



[https://erpscan.io/wp-content/uploads/2012/06/HITB-JIT-Spray-Attacks-and-Advanced-Shellcode.pdf]

Reusing Code

- Thinking like an attacker:
  - If we cannot inject our own code…
  - …can we **reuse code** that is already in **executable memory**?
  - **Linked libraries!**

Linked Libraries

- E.g. Google Chrome:
  - uses around 100 libraries
- The `libc` library is used by (almost) all programs, including functions like:
  - `system`: running a shell command
  - `mprotect`: changing memory permissions
- **Addresses of libraries are easier to predict**

Using `libc` as shellcode

- Use the techniques seen before…
- …but the return address will be that of a library function
- E.g., if we want to use `system` or `mprotect`, we have to **prepare the stack to simulate a genuine function call** to such functions:

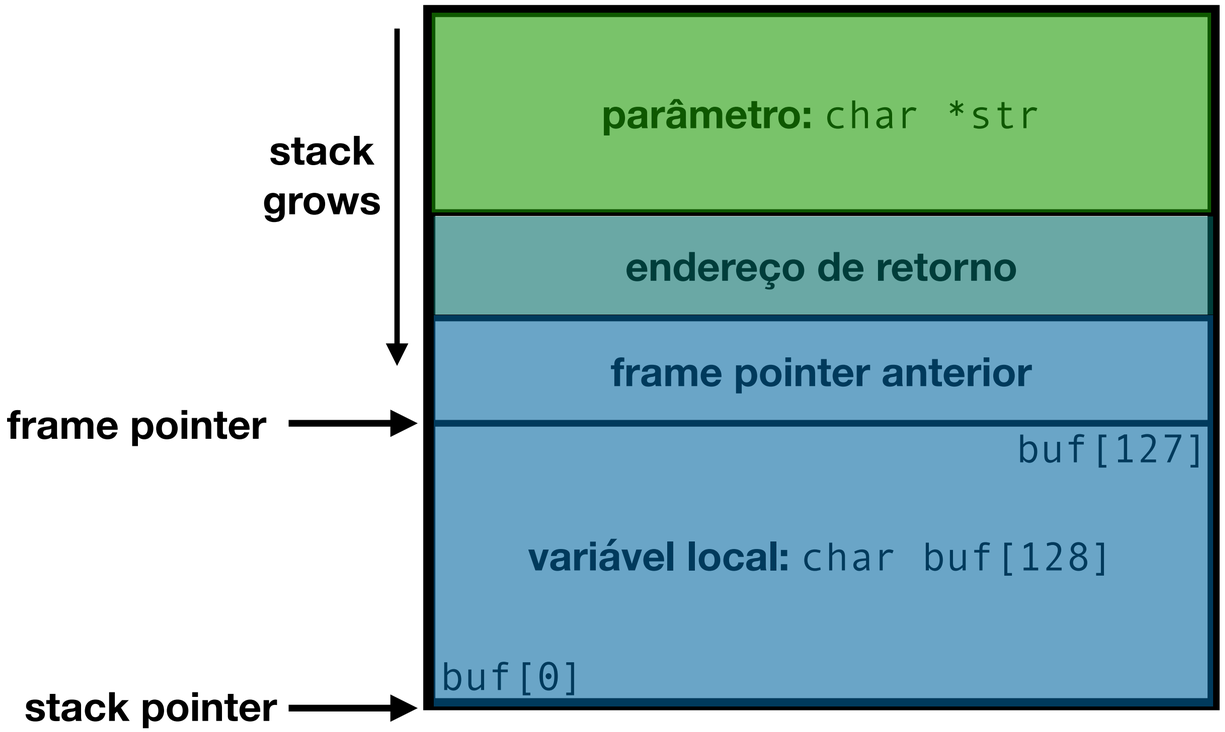
```
int system(const char *command);
int mprotect(void *addr, size_t len, int prot);
```

- In practice, requires understanding in detail how the stack works, we will only see simplified versions

Revisiting the Stack

- When entering `func`, its stack frame looks like:

```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    use(buf);
}
```

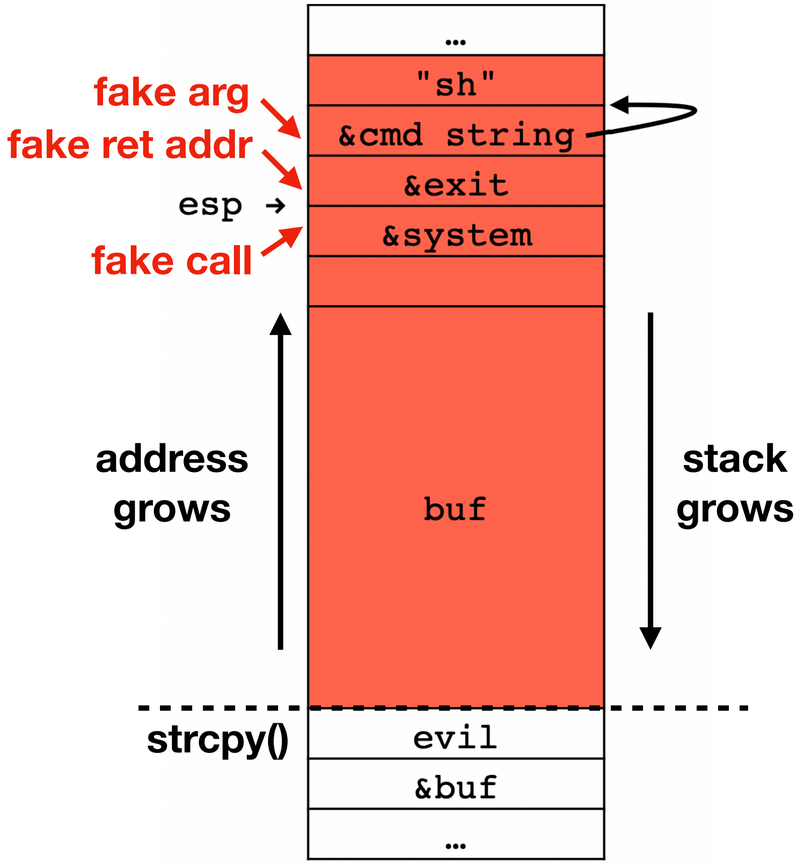


- The **calling function** stores parameters and the return address
- The **called function** stores the previous frame pointer and allocates its own local variables
- **Each function cleans up its used stack space**: the environment then uses the return address automatically, and the calling function cleans the parameters

How to call `system`?

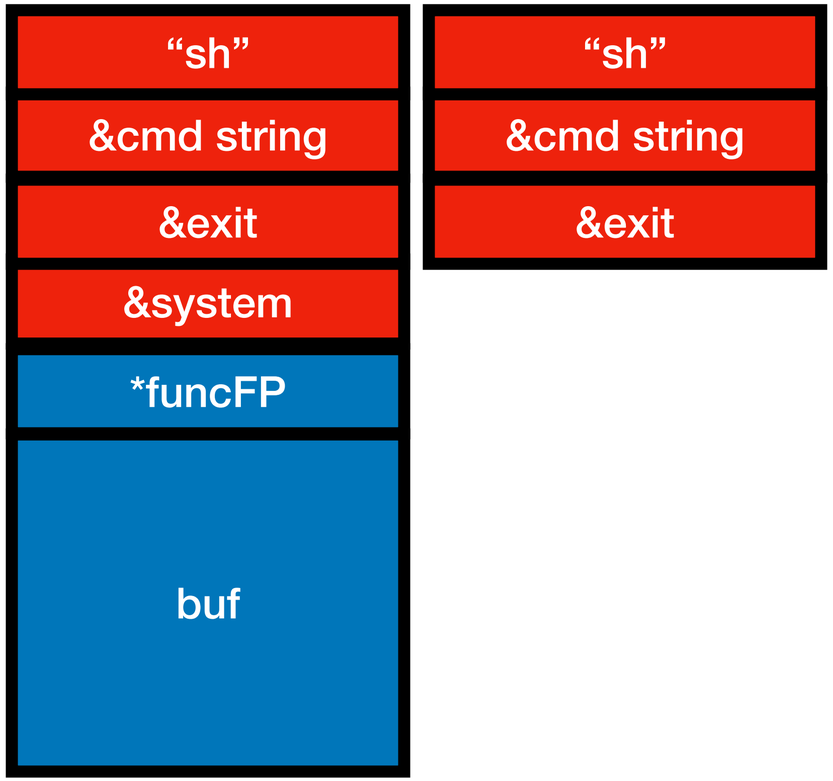
- The injected data has to:
  1. Replace the return address of the current function for `&system`
  2. Configure the stack for a valid call to `system`:
     - the address of a string with the invoked shell command (parameter)
     - a return address for when `system` exits
- To avoid a crash (why?): $\Longrightarrow$ avoiding detection
  - Place shell string far above the the actual frame
  - Return to function that ends program without errors, e.g., `exit()`

```
void foo(char *evil) {
    char buf[32];
    strcpy(buf, evil);
}
```



How to call `system`?

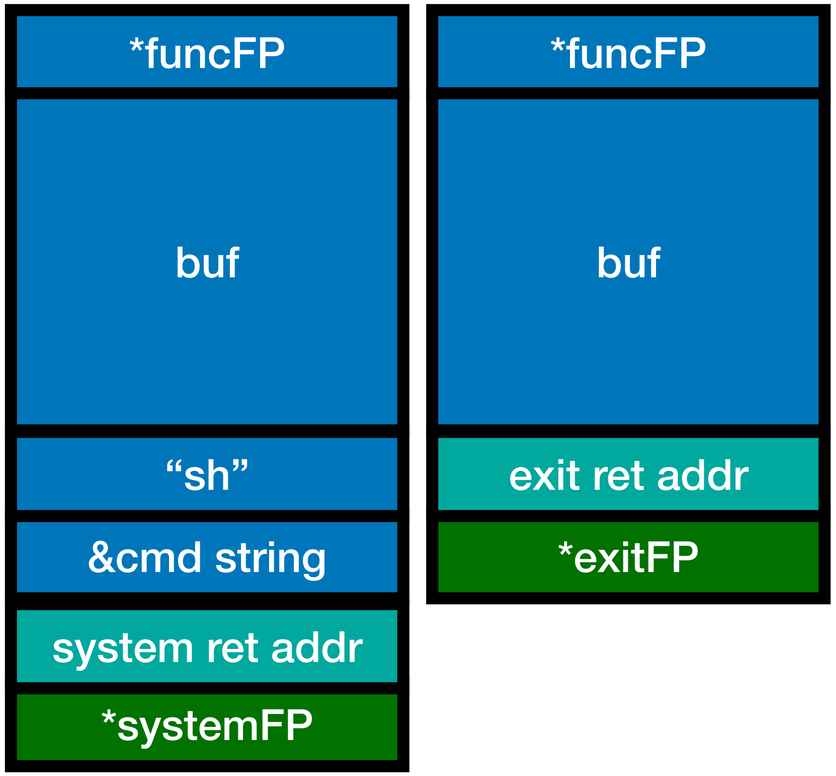
Fake Stack



Fake Code

```
void func(char *str) {
  char buf[128];
  system(“sh”);
  exit();
}
```

Real Stack

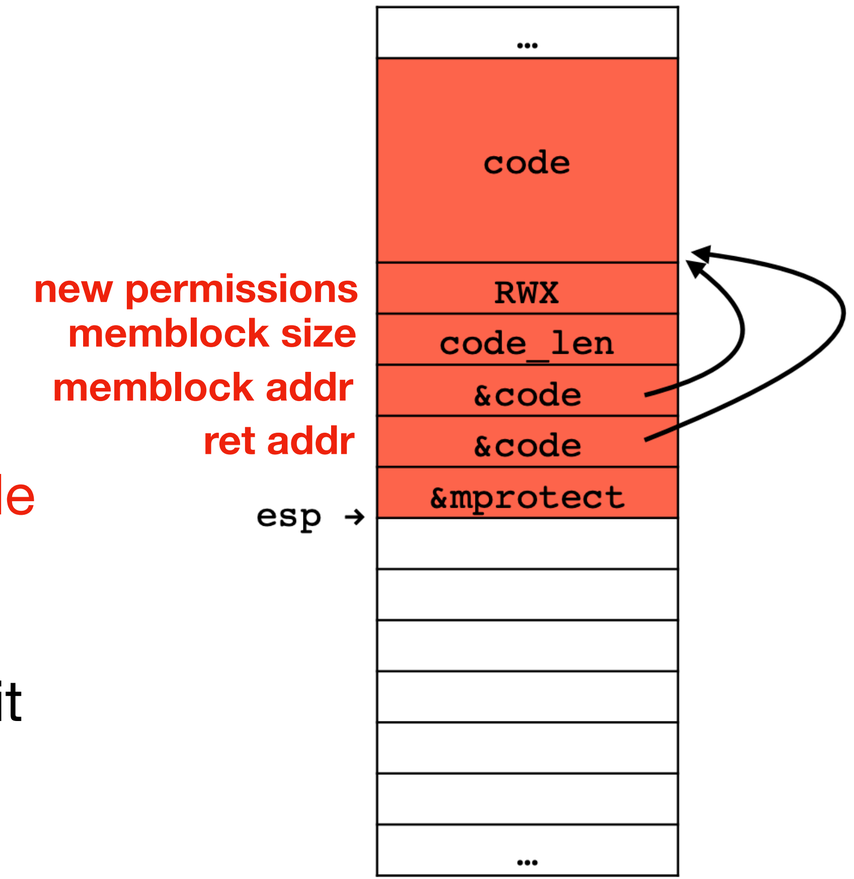


Where is the frame pointer for the fake `system` call?

How to call `mprotect`?

```
int mprotect(void *addr,
             size_t len,
             int prot);
```

- The injected data has to:
  1. Replace the return address of the current function for `&mprotect`
  2. Configure the stack for a valid call to `mprotect`:
     - the memory address of the injected shellcode, its size and its permissions (parameter)
     - a return address for when `mprotect` exits ⇒ shellcode
- Not easy because the `mprotect` parameters (addresses) generally contain `'\0'`, what reduces the changes of exploit (may not work with `strcpy` but with `memcpy`)

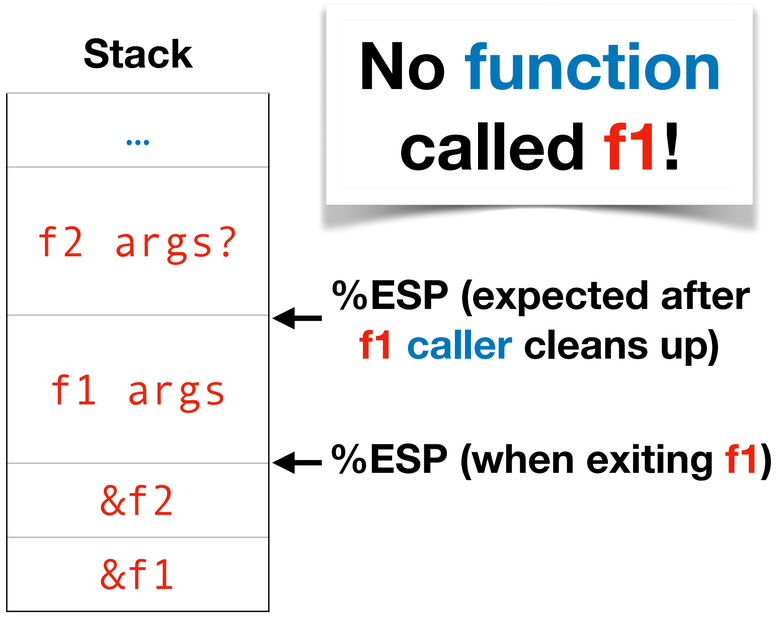


Reusing Code ⇒ ROP

- Thinking like an attacker:
  - If we can execute a library function…
  - …why not N functions in sequence?
  - **Return-Oriented Programming!**

ROP: complex but possible

- We need to recreate a stack structure that allows calling functions in sequence:
  - Return the first function to call
    - parameters + return address = second function to call



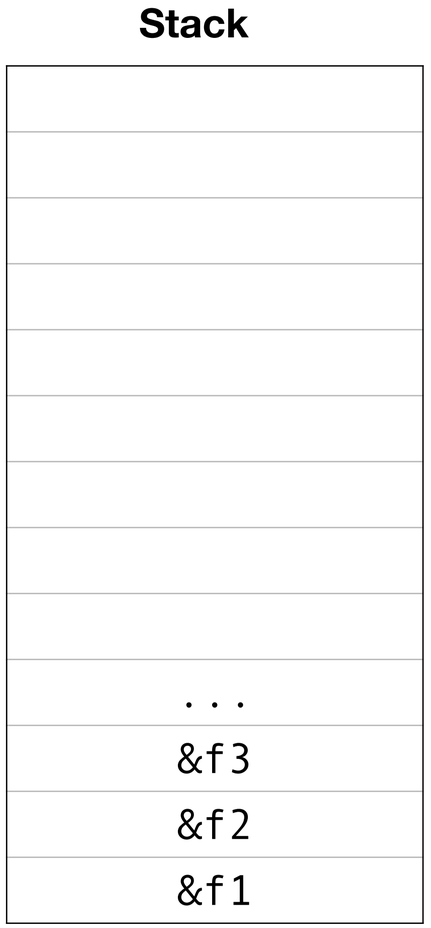
No function called f1!

- We placed in the stack parameters for the first function, but…
- The first function does not clean them!
- The second function is not responsible for cleaning parameters to itself or previous functions from the stack

- **When we enter the second function, the stack still has “garbage” from the previous call**

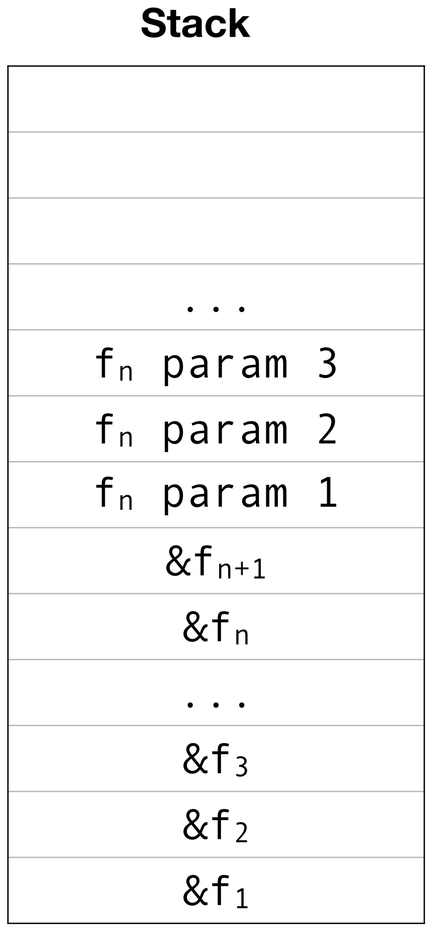
ROP: simple cases

- If a function does not receive parameters, the calling function stores only the return address in the stack and also cleans it!
  - Because the environment always clears the return address
- To execute **N functions without parameters**, we just place its addresses in reverse order in the stack
- `f1` returns to `f2`, which returns to `f3`, etc



ROP: simple cases

- After any sequence of n **functions that do not receive parameters**…
- …we can call **one function** f_n **that receives parameters**…
- …and end by calling **one function** f_{n+1} **that does not receive parameters**
- Why f_n **and not** f_{n+1}? The return address comes before the parameters in the stack

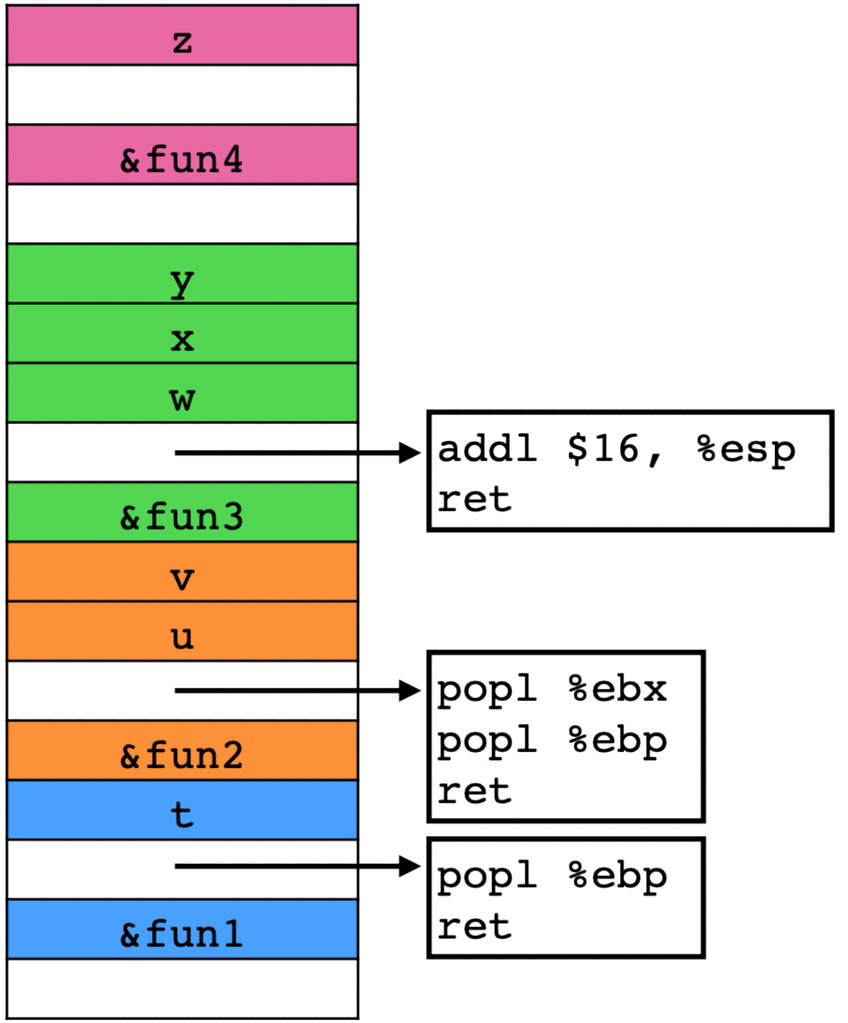


ROP: complex cases

- There are similar simple combinations, but what if we want to call **any sequence of functions with parameters**?

  `f1(t); f2(u,v); f3(w,x,y); …`

- We need to create alternate versions of such functions that clean their own stack parameters. How?
  - By placing another address in the stack between `f1` and `f2`
  - Such address must hold a piece of code that:
    1. `pops` the necessary number of bytes from the stack or directly moves the stack pointer `%esp`
    2. `returns` to the next function `f2`!



ROP in general

- Are the little sequences of code that we use to clean up the stack special?

  `pop <reg>; ret`

- In general geral, **if we find somewhere in executable memory**…

  `address1: i1; i2; i3; ret`

  `address2: i4; i5; i6; ret`

- …and place in the stack `address1` and `address2`, returning to `address1`…

- …we will execute `i1; i2; i3; i4; i5; i6; ret`

Return-Oriented Programming

- Look at the assembly code in memory (program or library) as a “letters soup”:

  1. Identify sequences (generally ending in `ret`, or more recently ending in `jmp` or `call`)

  2. Collect such sequences as reusable “gadgets”

  3. Join gadgets together

Image by Dino Dai Zovi

ROP: limits?

- Depends a lot on which code we have access to
- Arithmetic/logical operations and anything which is “straight-line” code (including shellcode) is simpler:
    1. Automatically search the target code for a set of gadgets
    2. Compose gadgets together such that they are **equivalent** to the desired behaviour
- Emulating conditionals/control-flow is more complex but possible (by playing with the stack pointer)
- Conceptually Turing-complete!

ROP history

- Beginning of times without DEP or ASLR ⇒  code injection
-  Post-DEP (2007) ⇒  `libc` gadgets
-  tools to automatize ROP
-  Post-ASLR ⇒  ROP using code that resides in predictable addresses
- **Nowadays**:  all OSes/compilers use position-independent executables (PIE) which can be loaded into random locations and make ROP much harder
-  tools for identification of gadgets at runtime!

Wrapping up Buffer Overflows

- If an attacker can **influence some input**:
  - It is vital to guarantee that **all memory accesses are correct**
- If an attacker can **write outside the designated memory region**:
  - Can subvert the control flow of the program
  - Can execute arbitrary malicious code
- If an attacker can only **read outside the designated memory region**:
  - Can read sensitive information
  - Can execute (potentially malicious) code residing in memory

Thousands of reported vulnerabilities

CVE List CNAs WGs Board About News

Search CVE List | Downloads | Data Feeds | Update a CVE Record | Request CVE IDs

TOTAL CVE Records: 240830

NOTICE: Transition to the all-new CVE website at WWW.CVE.ORG and CVE Record Format JSON are underway.

NOTICE: Support for the legacy CVE download formats ended on June 30, 2024.
New CVE List download format is available now on CVE.ORG.

HOME > CVE > SEARCH RESULTS

Search Results

There are **19123** CVE Records that match your search.

Name	Description
CVE-2024-9460	A vulnerability was found in Codezips Online Shopping Portal 1.0. It has been classified as critical. Affected is an unknown function of the file index.php. The manipulation of the argument username leads to sql injection. It is possible to launch the attack remotely. The exploit has been disclosed to the public and may be used.
CVE-2024-9407	A vulnerability exists in the bind-propagation option of the Dockerfile RUN --mount instruction. The system does not properly validate the input passed to this option allowing users to pass arbitrary parameters to the mount instruction. This issue can be exploited to mount sensitive directories from the host into a container during the build process and, in some cases, modify the contents of those mounted files. Even if SELinux is used, this vulnerability can bypass its protection by allowing the source directory to be relabeled to give the container access to host files.
CVE-2024-9355	A vulnerability was found in Golang FIPS OpenSSL. This flaw allows a malicious user to randomly cause an uninitialized buffer length variable with a zeroed buffer to be returned in FIPS mode. It may also be possible to force a false positive match between non-equal hashes when comparing a trusted computed hmac sum to an untrusted input sum if an attacker can send a zeroed buffer in place of a pre-computed sum. It is also possible to force a derived key to be all zeros instead of an unpredictable value. This may have follow-on implications for the Go TLS stack.
CVE-2024-9341	A flaw was found in Go. When FIPS mode is enabled on a system, container runtimes may incorrectly handle certain file paths due to improper validation in the containers/common Go library. This flaw allows an attacker to exploit symbolic links and trick the system into mounting sensitive host directories inside a container. This issue also allows attackers to access critical host files, bypassing the intended isolation between containers and the host system.
CVE-2024-9322	A vulnerability was found in code-projects Supply Chain Management 1.0. It has been classified as critical. Affected is an unknown function of the file /admin/edit_manufacturer.php. The manipulation of the argument id leads to sql injection. It is possible to launch the attack remotely. The exploit has been disclose to the public and may be used.
CVE-2024-9320	A vulnerability has been found in SourceCodester Online Timesheet App 1.0 and classified as problematic. This vulnerability affects unknown code of the file

How are vulnerabilities found?

- General pattern: memory management error ⇒ a program crash
- To search for vulnerabilities:
  - use a “fuzzer” to test the program against various (arbitrary) inputs
  - if it crashes, investigate why
- If the memory management error is confirmed, it is immediately recognised as a potential vulnerability
- Creating a working exploit typically requires much more work

A crash can be valuable!

Jailbreakers use Apple crash reports to 'free' iPhones

Bug hunt

Hackers like Mr Hill hunt for programming errors, or bugs, in Apple's software. Bugs may result in a program crashing or shutting down, and they are like gold dust to hackers because sometimes they can be exploited to create a jailbreak.

To help prevent this, Apple's phones record details of program crashes and send these reports back to the company. Apple's programmers can then analyse the crash reports and fix any underlying bugs that pose serious security risks or that could be exploited to create a jailbreak.

But crash reporting causes particular problems for Mr Hill and his team. That is because the hackers may have to crash a particular program thousands of times as they work out how to exploit a bug successfully, Mr Hill says, and this alerts Apple that the bug exists and that hackers may be investigating it.

In September Mr Hill was working on exploiting five separate bugs found in early versions of Apple's iOS 5 software to create a full or "untethered" jailbreak, but the most important ones had been patched by Apple when the final version of its software was released in October. Crash reporting was probably to blame, he believes.

Crash reports

The solution to this problem is to subvert Apple's crash reporting capability by turning it against the company, he says.

"Chronic Dev is ready to turn this little information battle into an all-out, no-holds-barred information WAR," Mr Hill wrote on the Chronic-Dev blog recently, using his nom de guerre Posixninja.

To do this he has written and distributed a program called CDevreporter that iPhone users can download to their PC or Mac. The program intercepts crash reports from their phones destined for Apple and sends them to the Chronic Dev team.

If crash reports are like gold dust then Mr Hill and his team are now sitting on a gold mine.

"In the first couple of days after we released CDevreporter we received about twelve million crash reports," he says.

"I can open up a crash report and pretty much tell if it will be useful or not for developing a jailbreak, but we have so many that I am working on an automated system to help me analyse them."

https://www.bbc.com/news/technology-16204663

Ethical Hacking

- It is crucial to:
  - Have experts searching for vulnerabilities
  - be very careful about how vulnerabilities are disclosed
- The bug bounty and vulnerability reporting platforms are designed to avoid reported “0-days” ending up causing damage
- Are we done after the bug is reported?

Google Security Blog

The Ups and Downs of 0-days: A Year in Review of 0-days Exploited In-the-Wild in 2022

July 27, 2023

Over 40% of the 0-days discovered were variants of previously reported vulnerabilities. 17 out of the 41 in-the-wild 0-days from 2022 are variants of previously reported vulnerabilities. This continues the unpleasant trend that we've discussed previously in both the 2020 Year in Review report and the mid-way through 2022 report. More than 20% are variants of previous in-the-wild 0-days from 2021 and 2020.

"At least half of the 0-days we've seen in the first six months of 2022 could have been prevented with more comprehensive patching and regression tests.

https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html

SQLSlammer (2002)

- Buffer overflow identified (and fixed) in SQL Server
- A security researcher publicly demonstrated (blog “writeup”) how to exploit this bug to propagate a worm
- Basis of the code for the SQLSlammer worm
- **A vast number of systems hadn’t been patched**

The spread of SQL Slammer

At its height, SQL Slammer, which was the most widespread worm since 2001’s Code Red worm, doubled in size every 8.5 seconds. South Korea, one of the most connected countries in the world at the time, had an outage of internet and cell phone coverage for 27 million people, while in the US, almost all of Bank of America’s 13,000 ATMs were temporarily knocked offline.

https://www.welivesecurity.com/2016/09/30/flashback-friday-sql-slammer/
https://en.wikipedia.org/wiki/SQL_Slammer

Blaster worm (2003)

Blaster (also known as **Lovsan**, **Lovesan**, or **MSBlast**) was a computer worm that spread on computers running operating systems Windows XP and Windows 2000 during August 2003.[1]

The worm was first noticed and started spreading on August 11, 2003. The rate that it spread increased until the number of infections peaked on August 13, 2003. Once a network (such as a company or university) was infected, it spread more quickly within the network because firewalls typically did not prevent internal machines from using a certain port.[2] Filtering by ISPs and widespread publicity about the worm curbed the spread of Blaster.

Creation and effects [edit]

According to court papers, the original Blaster was created after security researchers from the Chinese group Xfocus reverse engineered the original Microsoft patch that allowed for execution of the attack.[4]

The worm spreads by exploiting a buffer overflow discovered by the Polish security research group Last Stage of Delirium[5] in the DCOM RPC service on the affected operating systems, for which a patch had been released one month earlier in MS03-026 and later in MS03-039. This allowed the worm to spread without users opening attachments simply by spamming itself to large numbers of random IP addresses. Four versions have been detected in the wild.[6] These are the most well-known exploits of the original flaw in RPC, but there were in fact another 12 different vulnerabilities that did not see as much media attention.[7]

How to avoid these attacks?

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: Low-level Software Security III: Integer overflow, ROP and CFI
  - CS155: Control Hijacking: Defenses
  - CS343: Code reuse attacks + Return-oriented programming