Software Exploitation

BASICS TO MODERN

Lecture Parts

01

Basics To Modern

02

Heap Exploitation

03

Other Vulnerabilities

Plan

01 ASM Reminder

02 Tools

03 Shellcode

04 Basic Stack Buffer Overflow

05 Format String

06 ASLR, PIE & ROP

Plan

01

ASM Reminder

What is assembly ?

Assembly

- Human representation of the instructions.
- 1:1 machine code translation.
- Assembler makes the translations.
- Specific to each CPU.
- Assembly allows to give orders to the CPU.
  - It is NOT made for writing algorithms!

Do all CPUs have the same ASM ?

Assembly x86

- 3 modes exist:
  - 16 bits (1978)
  - 32 bits (1985)
  - 64 bits (2001/2003): x86_64 or x64
- Complex instruction set computer (CISC) architecture.
- Most used architecture for servers and computers.
- 2 syntaxes:
  - AT&T
  - **Intel**

Mnemonic & Opcode

- Actual order for the CPU: tells the CPU what it should do.
- Different kinds of instruction:
  - Arithmetic
  - Logic
  - Memory access
  - Control Flow
  - …
- A mnemonic is the human representation of an opcode.
- Opcodes are the bytes representation of the mnemonic.

Assembly x86: instructions

PREFIX MNEMONIC_OPCODE OPERAND_DST, OPERAND_SRC

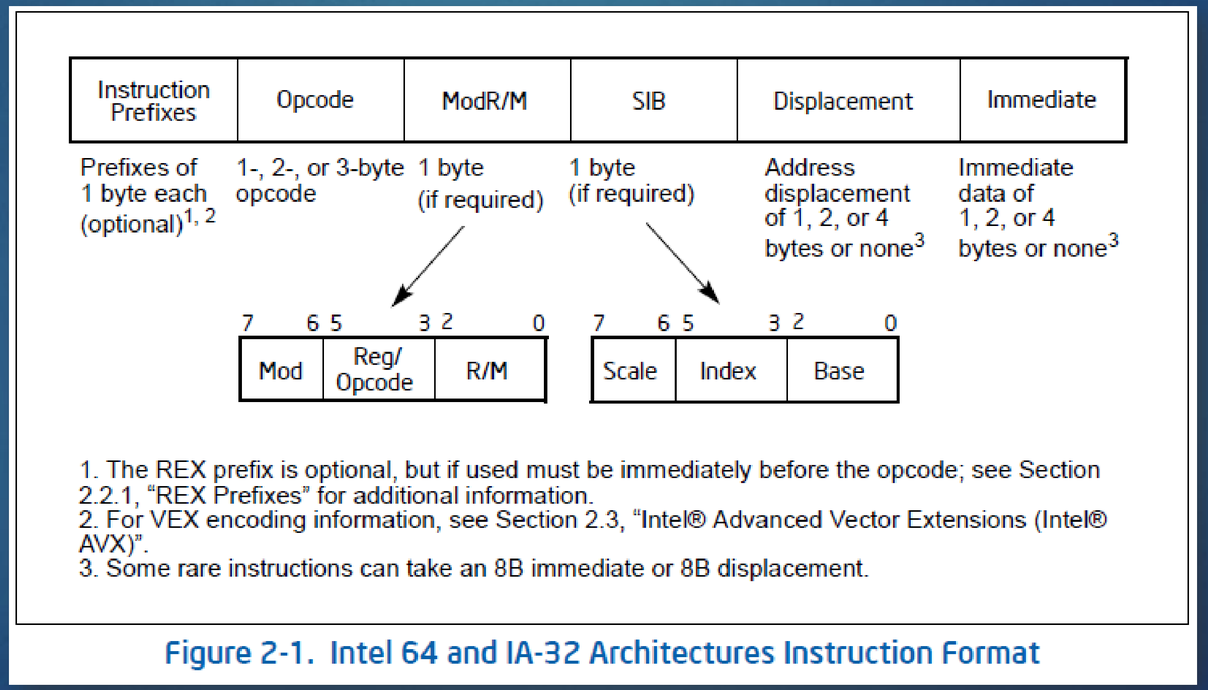
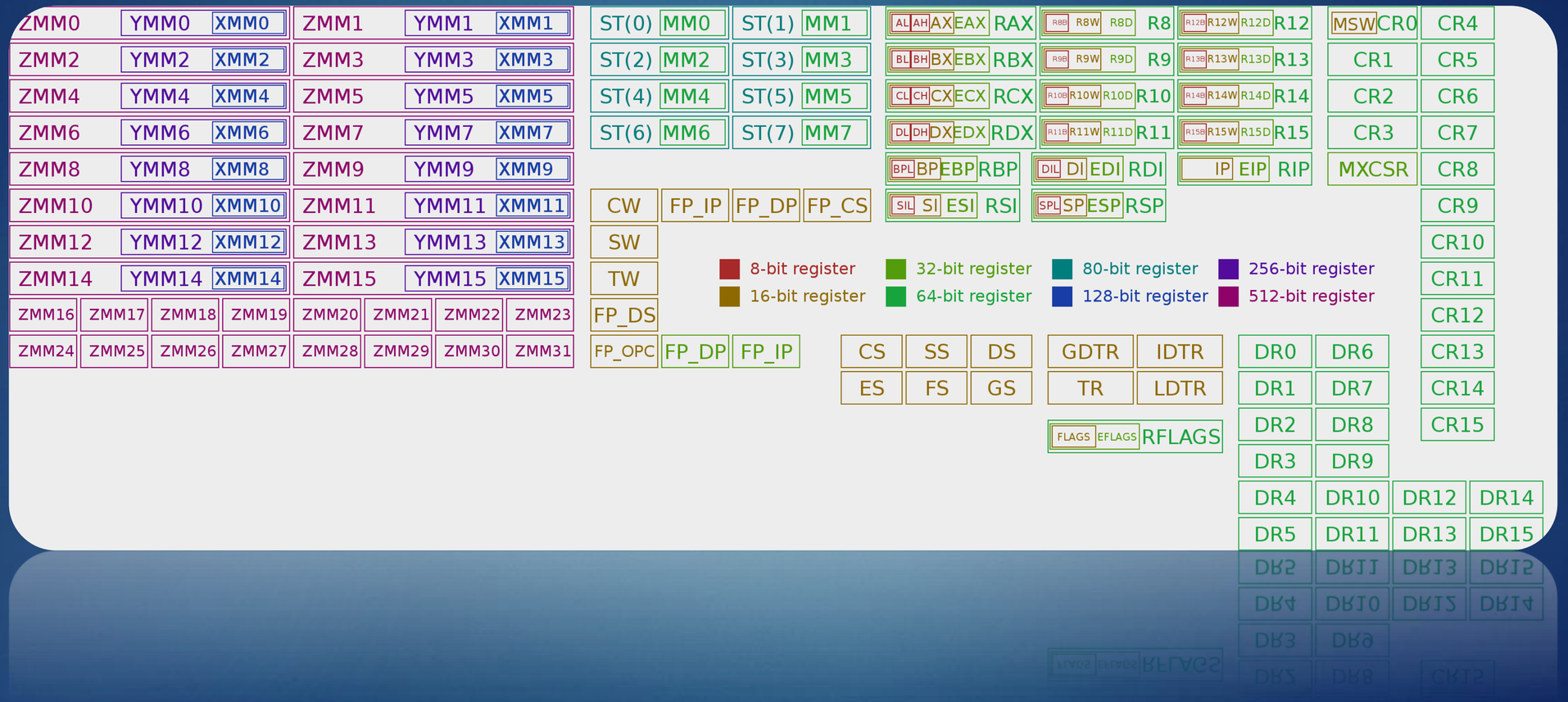


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Assembly x86: registers



Basic instructions

Assembly	Equivalent
MOV RAX, 8	RAX = 8
MOV [RAX], 8	*RAX = 8
MOV RBX, RAX	RBX = RAX
ADD RBX, RAX	RBX += RAX
ADD RAX, RAX	RAX += RAX

Calling Convention ?

Calling Convention

- Just a convention.
- Several of them.
  - Potentially several in the same binary!
- Linux 64bits: rdi, rsi, rdx, rcx, r8, r9, <stack>
- Linux 32bits: **?**

- https://en.wikipedia.org/wiki/X86_calling_conventions
- Part of the Application Binary Interface (ABI)
  - Also: exceptions, mangling, …

Calling Convention

- Just a convention.
- Several of them.
  - Potentially several in the same binary!
- Linux 64bits: rdi, rsi, rdx, rcx, r8, r9, <stack>
- Linux 32bits: <stack>
- https://en.wikipedia.org/wiki/X86_calling_conventions
- Part of the Application Binary Interface (ABI)
  - Also: exceptions, mangling, …

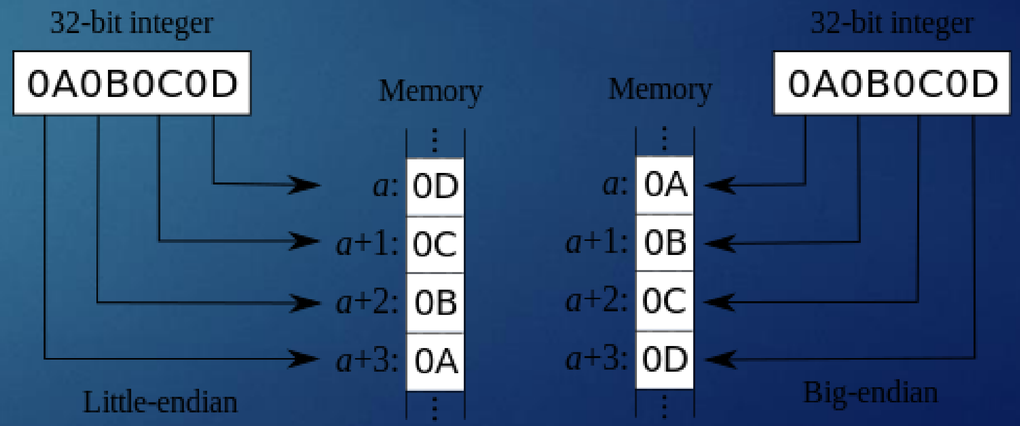
Endianness ?

Endianness

> Endianness is the order or sequence of bytes of a word of digital data in computer memory.
>
> Wikipedia

Number: 0xABCD6789

- Little Endian: **?**
- Big Endian: **?**

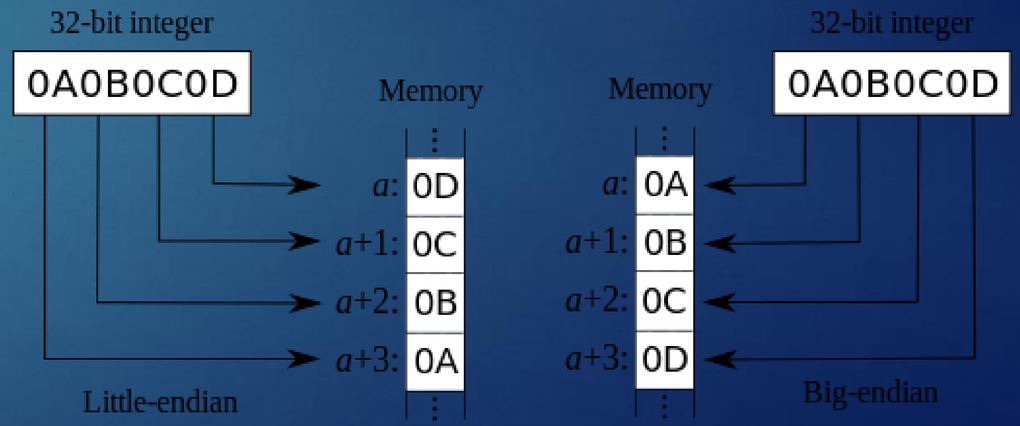


Endianness

> Endianness is the order or sequence of bytes of a word of digital data in computer memory.
>
> Wikipedia

Number: 0xABCD6789

- Little Endian: \x89\x67\xCD\xAB
- Big Endian: \xAB\xCD\x67\x89



Wait! How do we translate that to machine code ?

Reminder: Compilation

01 **Preprocessing**
Resolve macro, include headers, …

02 **Compilation**
Convert to assembly or low-level intermediary language.

03 **Assemble**
Convert assembly to machine code.

04 **Linking**
Link the pieces of machine code together and with static library.

Where is all of that documented ?

Intel Manual

- Intel® 64 and IA-32 Architectures Software Developer Manuals
  - 4 basic volumes
  - ~5000 pages total

- Volume 1: basic of the processor (register, memory, …)
- Volume 2: all the instructions supported
- Volume 3: system programing
- Volume 4: Model Specific Register (part of system configuration)

- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Plan

01

ASM Reminder

02

Tools

NASM

"Netwide Assembler (NASM), an asssembler for the x86 CPU architecture portable to nearly every modern platform, and with code generation for many platforms old and new."

- Simple to use.
- Works on Windows and Linux.
- Man page available

```
nasm -f elf64 myhello.S # produce myhello.o
gcc -no-pie myhello.o -o myhello

nasm -f bin -o shell shell.S # produce directly the binary
```

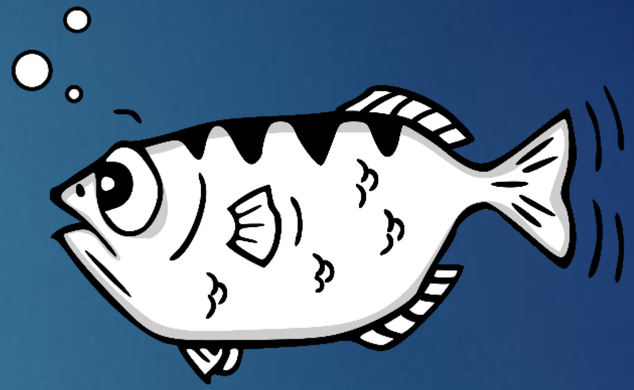
Disclaimer!

Source code is **NOT** provided for all exercises!

Tools: Reversing

Tools: debugger

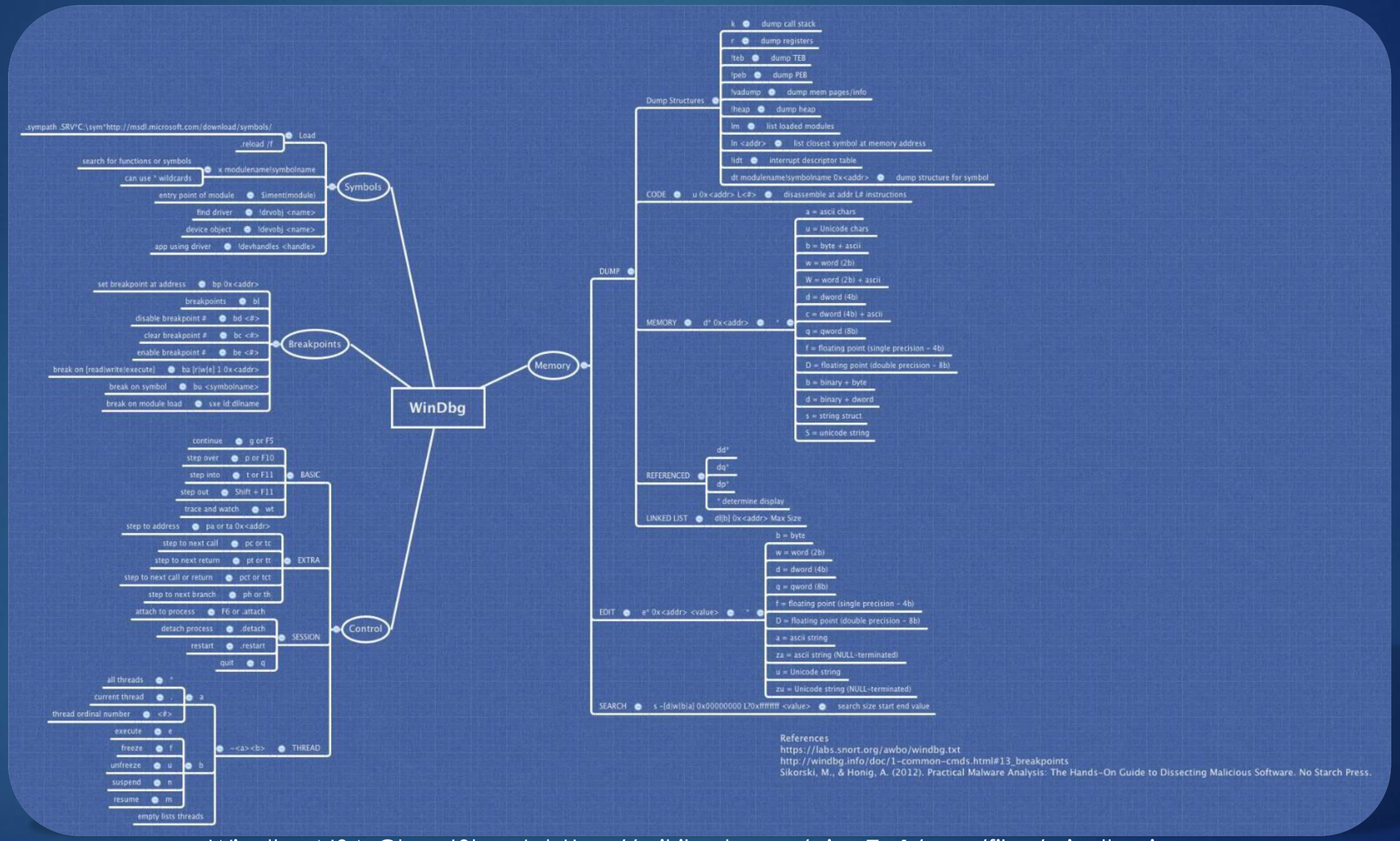
- **Gdb & windbg**
- Other tools:
  - IDA/Ghidra integration
  - Ollydbg
  - X64dbg
  - …
- Debug information ?



Debugging with gdb

- Setup intel syntax (default AT&T):
  - **set disassembly-flavor intel**
- Breakpoint at an address:
  - **b *ADDR**: (opposite: **delete**)
- Control execution:
  - **continue:** (**c**): continue
  - **si**: *step* one instruction (go into call)
  - **ni**: *next* one instruction (do not go into call)
- View instruction:
  - **x/5i ADDR**: print 5 instruction at address
  - **display/5i ADDR**: print each time we stop (**undisplay** for the opposite; **display/5i $rip**)
- View register:
  - **info register**: (**i r**): print values of all registers
  - **i r rax**: print value of RAX
- View call stack:
  - **backtrace**: (**bt**): print the stack trace
- View mapping:
  - **info proc map**: mapping with address
- Follow fork:
  - **set follow-fork-mode child**
- More *GDB Quick Reference*: https://www.cs.utexas.edu/~dahlin/Classes/UGOS/reading/gdb-ref.pdf

Debugging with windbg



Windbg NSA CheatSheet: https://wikileaks.org/ciav7p1/cms/files/windbg.jpg

Scripting with pwntools

CTF Framework & exploit development library, written in Python.

https://docs.pwntools.com/en/stable/index.html

Installation: `pip install pwn --user`

```
from pwn import *

# CONNECT:
target = remote("example.com", 9000) # TCP connection on port 9000
target = process("./exo100") # launch exercise exo100

# SEND & RECV:
target.send("hello") # send hello to target
target.sendline("hello") # send hello\n to target

print(target.recvline()) # recv a line from target
print(target.recvuntil("world")) # recv until "world" is read

# PACK & UNPACK:
p32(0xAABBCCDD) # pack little endian 32bits
u32(b"\xDD\xCC\xBB\xAA") # unpack little endian 32 bits
p64(0xAABBCCDDEEFF9988) # pack little endian 64bits
u64(b"\x88\x99\xFF\xEE\xDD\xCC\xBB\xAA") # unpack little endian 64 bits

# FINAL EXAMPLE:
target.send(p64(0xAABBCCDD))
target.interactive() # direct interaction with the target!
```

Plan

01

ASM Reminder

02

Tools

03

Shellcode

What are we going to inject when using a memory corruption ?

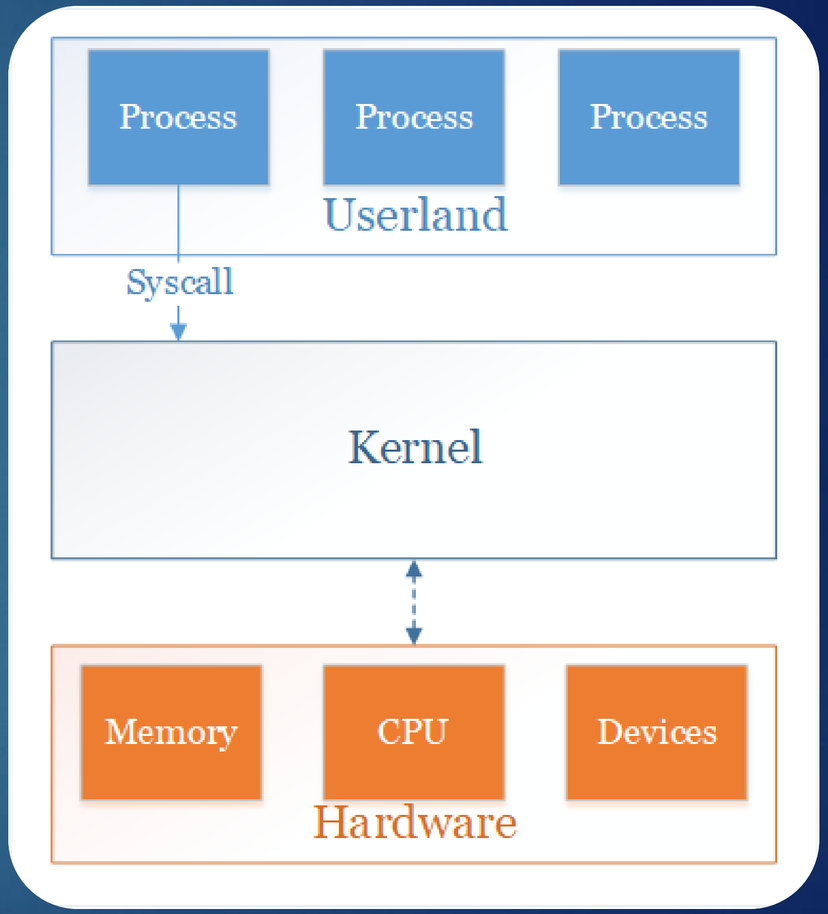
Shellcode

- "A shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability." (Wikipedia)
- Called shellcode because the usual goal is to get a shell.
  - But you can have whatever payload you want.
- Usually written directly in ASM because depends on the vulnerability.
- In general it is the final step of exploitation.
- Triggering the vulnerability allows you to ”jump” on your shellcode.

What are syscalls ?
Why do you want to use them ?

Syscall

- System call
  - Call to the kernel/system
- Special instruction(s) for doing this
  - x86: int 80h
  - x64: syscall
- Different syscall identified by the syscall number.



Syscall convention

- Special calling convention, for x64:
  - rax: syscall number
  - argument: rdi, rsi, rdx, r10, r8, r9
  - *man 2 syscall* will give you those information.

- The syscall numbers
  - Depends on the architecture & the OS
  - Can be found in (kernel) headers (/usr/include/asm/unistd_64.h)
  - Or online: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

Practice!

Shellcode

Information:

- shell100 & shell101 on CTFd
  - Careful to architecture!
- Launch them first on your computer!
  - strace & gdb for debugging.
- Use pwntool or a python script for writing the solution.

How could we stop the execution of shellcode ?

Data Execution Prevention (DEP)

- Data Execution Prevention (NX, W^X, ...)
- Basic idea is Write xor Execute
  - You can't have both at the same time
- You can't execute code on your stack, heap...

If we can't inject our code, what are we going to do **?**

Plan

01 ASM Reminder

02 Tools

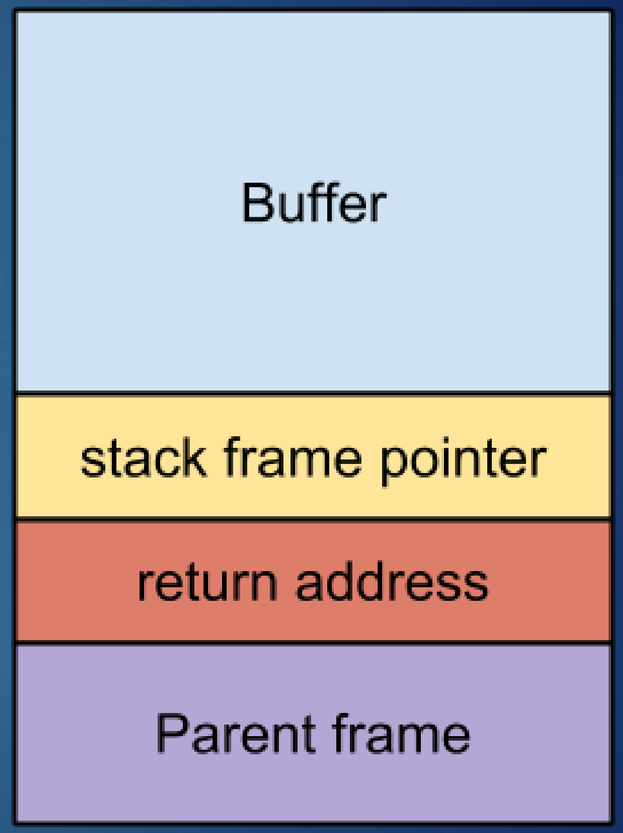
03 Shellcode

04 Basic Stack Buffer Overflow

What is a buffer overflow ?
What kind there is ?

The Stack ?

- A memory zone (like the heap)
- Used for:
  - Giving the arguments
    - Dependent of the calling convention.
    - For x86 (32bits) usually on the stack
  - Local variables.
  - **Return address of the functions.**



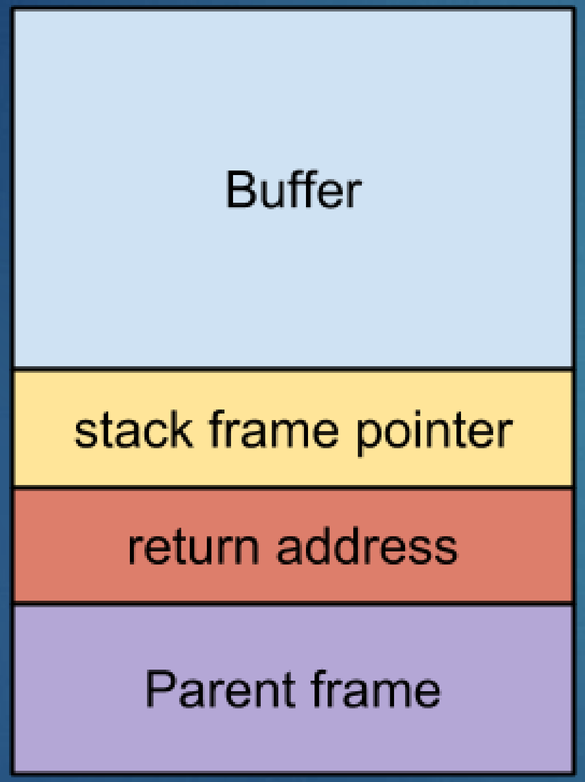
A first BOF

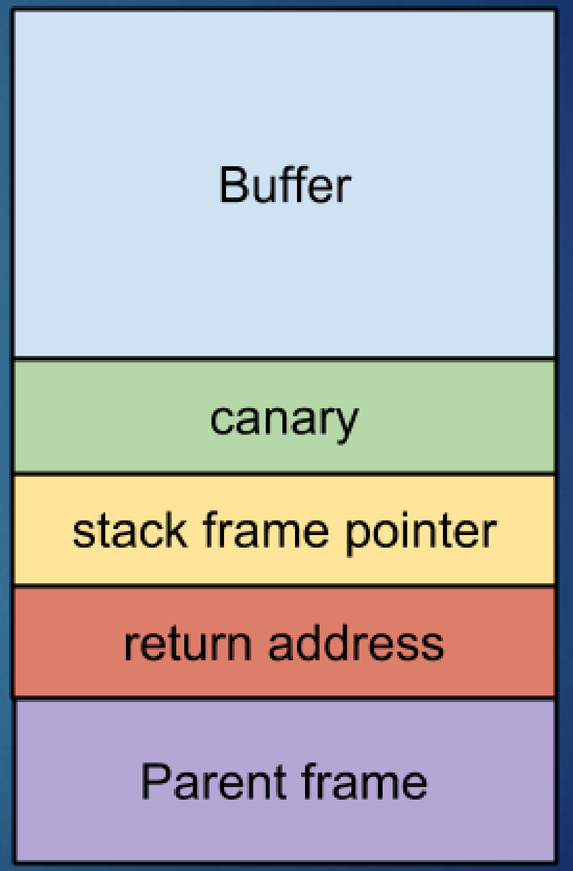
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
    printf("you have changed 'modified'\n");
  } else {
    printf("Try again?\n");
  }
}
```

Stack cookies

- Stack cookies/stack canaries are mitigation for avoiding stack based buffer overflow.
- They are basically a magic value which:
  - Is initialize at the start of the execution of a program.
  - Is put on the stack at the beginning of execution of a function.
  - Should stay the same during the execution of the function.
  - Is check at the end of the function.
- If the value of the stack cookie has been corrupted:
  - a buffer overflow occurred.
  - kill the program.

Canary





Who sets the canary ?

How do we bypass the canary ?

Canary bypass

01

Leak its value

Being able to read the value. The most common one!

02

Bad canary

Bad entropy for the canary or always the same.

03

Brute force

If possible 1 bytes at a time. Or in specific case with bad entropy.

04

Corrupt before it

The canary will not protect data before the return value: other lvar.

Practice

New CTF exercises!

Plan

01 ASM Reminder

02 Tools

03 Shellcode

04 Basic Stack Buffer Overflow

05 Format String

Format String ?

> Code such as printf(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain %n, causing the printf() call to write to memory and creating a security hole.
>
> man 3 printf (BUGS)

- Variadic number of arguments in printf (va_args).
- How does the code knows the number of arguments for printf ?

Format String ?

> Code such as printf(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain %n, causing the printf() call to write to memory and creating a security hole.
>
> man 3 printf (BUGS)

- Variadic number of arguments in printf (va_args).

- How does the code knows the number of arguments for printf ?
  - Count the number of % in the string.
  - What happens if we put more % than there is actual arguments ?

Exploiting Format String

- We can obviously leak data…
- What about %n **?**

*The number of characters written so far is stored into the integer specified by the int * (or variant) pointer argument. No argument is converted.*

```
int i;
printf("%s%n", "Hello", &i); // i == 5
```

- Write to the address of a given argument (&i) the number we want to write. (what)
- At one point the arguments are taken from the stack (va_arg).
  - If the buffer was once on the stack: we can take the content of the buffer as argument. (where)
- Set the address to write in the buffer and take this one as argument for the %n: write-what-where!

Format options

Format	Description
%[num]$[option]	Takes the arg. num for option (%2$x draw the hex of the second arg.)
%.[num][option]	Draws at least num byte (actually depends of the given option) (%.200x: draw the first arg with at least 200 chars).
%n	Write an int (4 bytes) at the address.
%hn	Write a short (2 bytes) at the address.
%hhn	Write a byte (1 byte) at the address.

Write-What-Where ?

- write-what-where:
  - **write what** we want **where** we want,
  - almost always equivalent to success.

What can we possibly rewrite **?**

Write-What-Where ?

- write-what-where:
  - **write what** we want **where** we want,
  - almost always equivalent to success.

- Rewrite:
  - A stack return? good if we don't have ASLR, but what if we have some?
  - A pointer to function? If we have one, and know where it is.
  - The GOT? Almost always one, not affected by ASLR.
    - But if we have PIE will need a leak.

GOT ? PLT ?

Reminder: Execution

01 **Read file format**
Kernel or interpreter.

02 **Map in memory**
Kernel or interpreter.

03 **Link & load**
Dependencies handling.
Kernel, linker or interpreter.

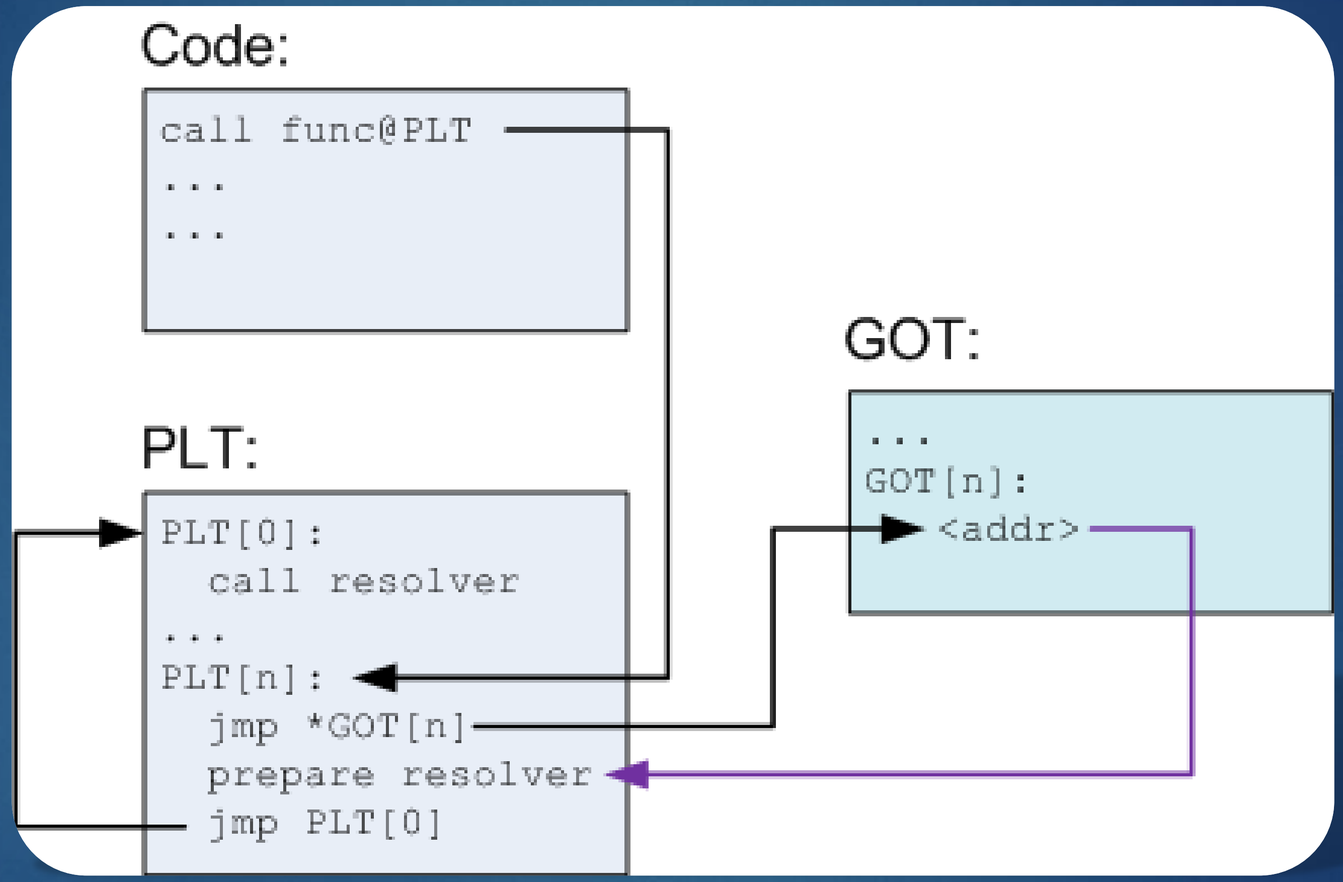
04 **Call entry point**
This is **not** the main function.

GOT & PLT

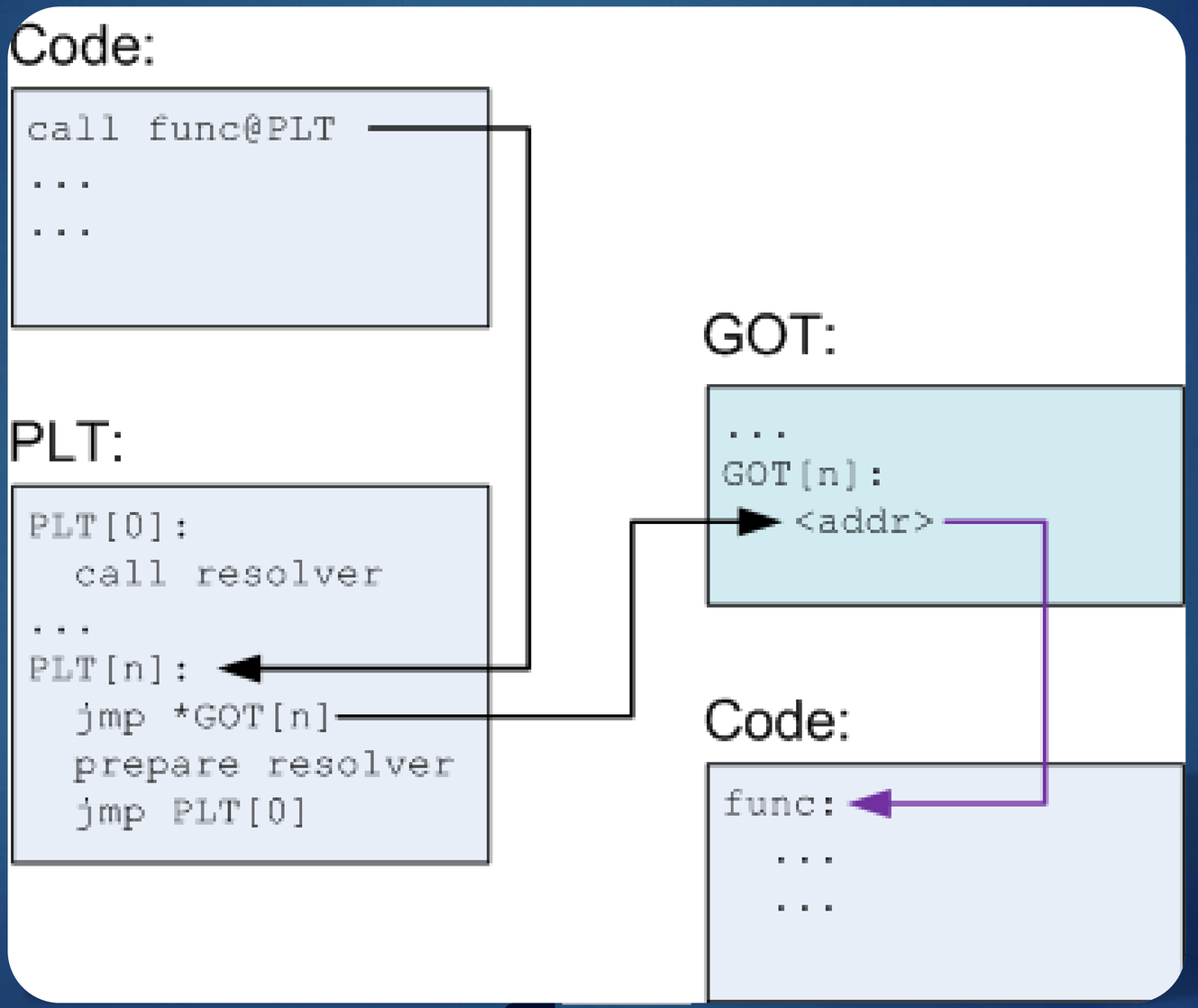
- The GOT & PLT are sections.
- Global Offset Table (GOT):
  - Array of addresses of the external functions.
- Procedure Linkage Table (PLT):
  - Code which is in charge of calling the functions.

- The dynamic linker needs other information (in other sections) for making the link in both the source and target binaries
  - .dynamic, .dynsym, .dynstr, .strtab, .symtab
- This is done in *lazy*.

GOT & PLT



GOT & PLT



Practice

New CTF exercises!

Does format string vulnerabilities occurs only with printf **?**

Plan

01 ASM Reminder

02 Tools

03 Shellcode

04 Basic Stack Buffer Overflow

05 Format String

06 ASLR, PIE & ROP

ASLR & PIE

What is ASLR ?

ASLR

- Address space layout randomization (ASLR)
- If not enabled, everything is always at the same address (the stack, the heap, the library. . .)
- When enabled the base address of the stack, the heap and the libraries are randomized.
- But the address of the loaded binary is **not**.

```
echo 1 > /proc/sys/kernel/randomize_va_space
```

What problems create ASLR for the exploiter ?

How can we bypass ASLR ?

PIE

- Position-independent executable (PIE)
- Like ASLR but with the base of the binary randomized.
- -fpie for gcc, -pie for ld

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

- People will often speak about “ASLR” for saying ASLR+PIE

How can we bypass PIE ?

Bypassing ASLR & PIE

- We need an address leak.
- Once we have the leak, we can calculate the position of the base address:

```
(leak_addr - (addr_not_pie - base_addr_not_pie)
```

- Once we have the base address we can't get any address of the binary.
  - Simple once we have a leak...

ROP

ROP

- Return Oriented Programming. (ROP)
- The basic idea is to create the whole stack and use returns to call the parts of the existing code.
- Because of DEP it is not possible to inject code and get it executed.
- This technique allows us to re-use the code of the binary to do what we want.
- In x86 the ROP is simpler because the calling convention use the stack, in x86_64 the calling convention use registers.

Gadgets

- In order to use ROP:
  - we need sequences of machine instructions ending by "ret" (or something like "call [eax]").
  - Those pieces of code are called gadgets.

- A typical gadget is something like:

```
pop [REG] ; pop [REG] ; ret
```

- For finding them you can use tools:
  - rp++: https://github.com/0vercl0k/rp
  - ROPGadget: https://github.com/JonathanSalwan/ROPgadget
  - Ropper: https://github.com/sashs/Ropper
  - …

Are instructions aligned in x86 ? What happens if we jump in the middle of an instruction ?

Question!

We want to create a **x64 Linux** ROP chain for calling:

1. *func1* with 2 arguments.
2. and then *func2* with 1 argument.

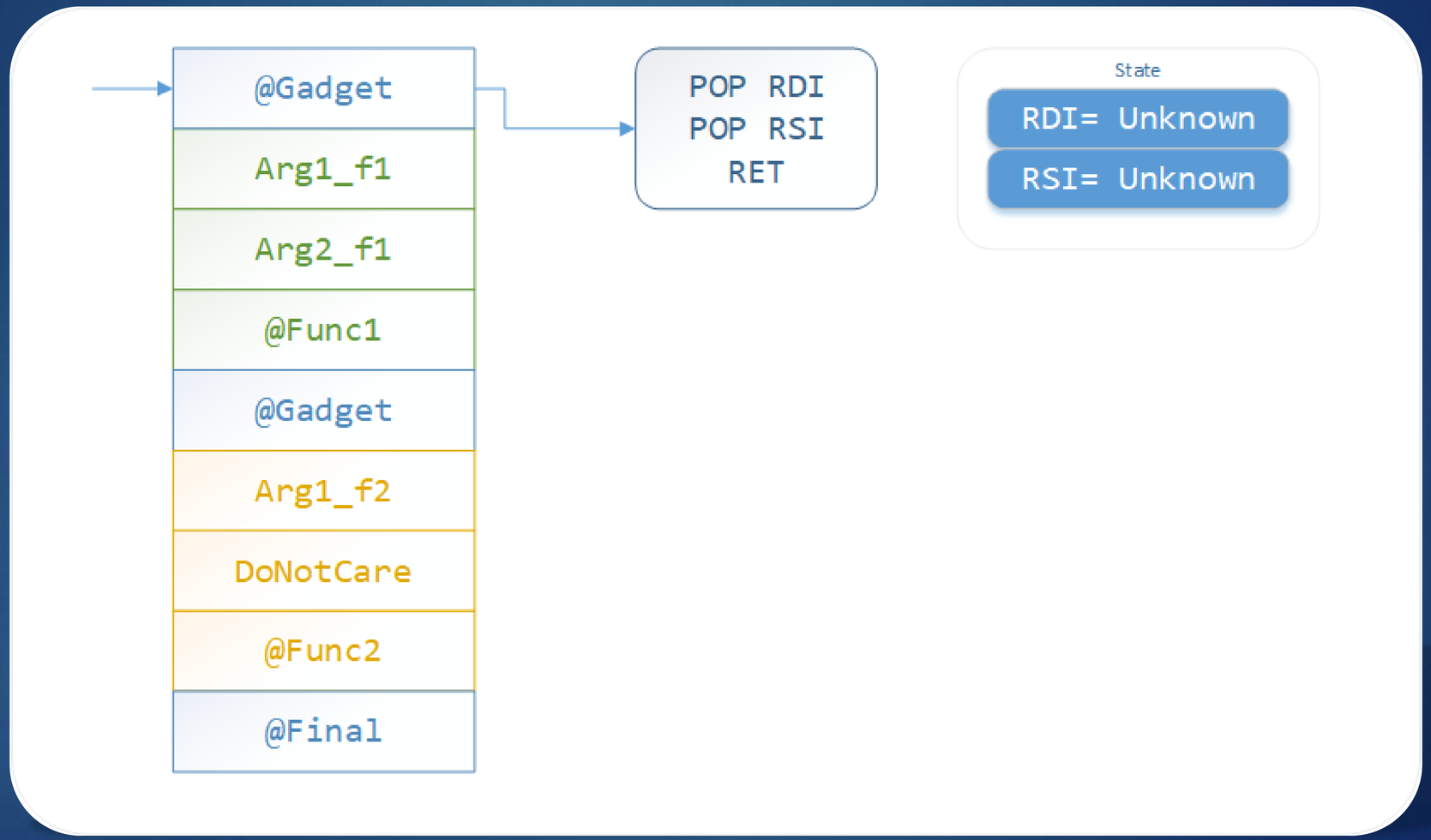
We have:

- A gadget: `pop RDI ; pop RSI ; ret`
- A stack BOF allowing us to rewrite anything.

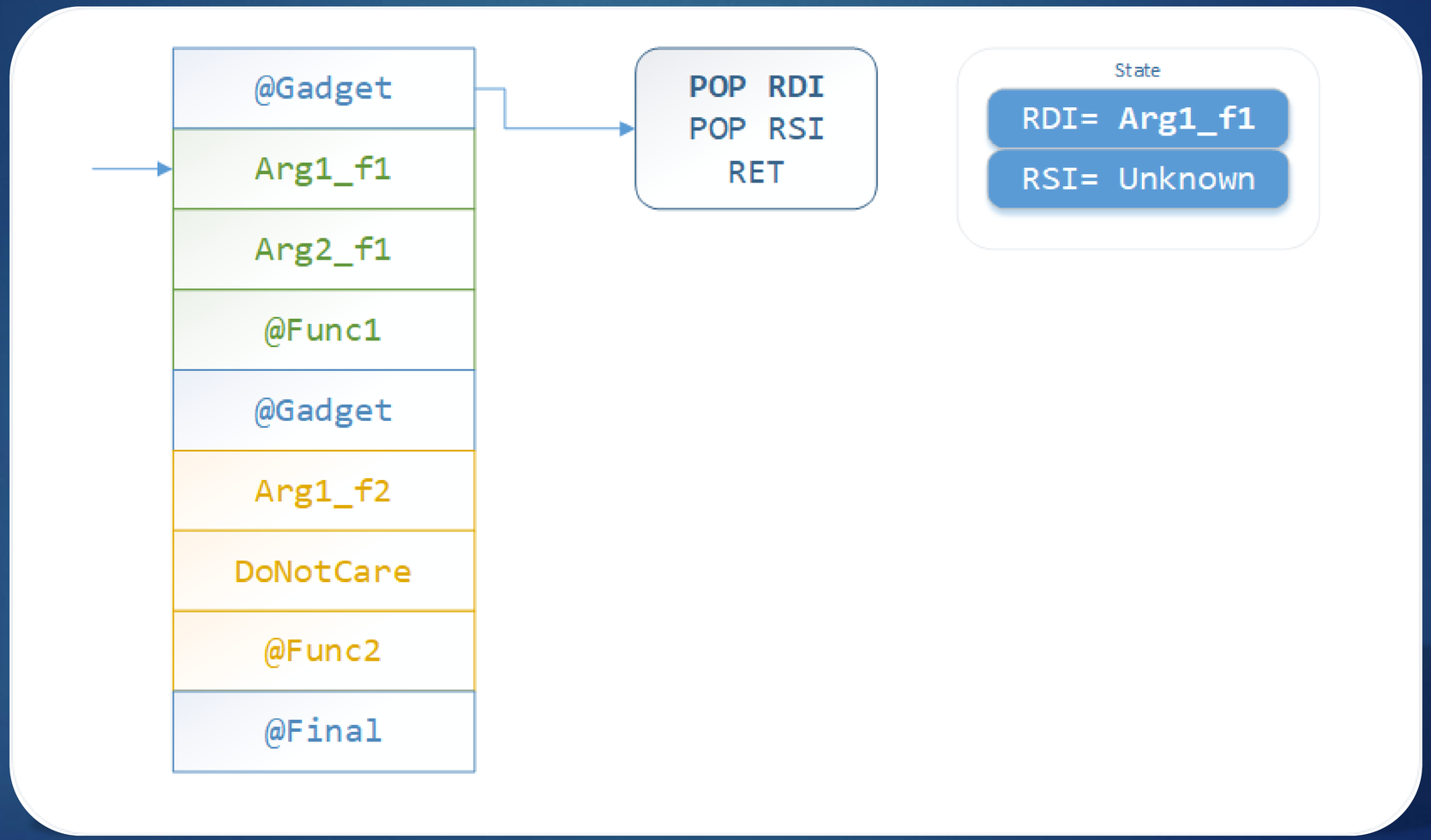
Questions:

1. What is the calling convention in x64 for Linux **?**
2. What is our created stack going to look like **?**
3. Same question for x86 32bits**?**

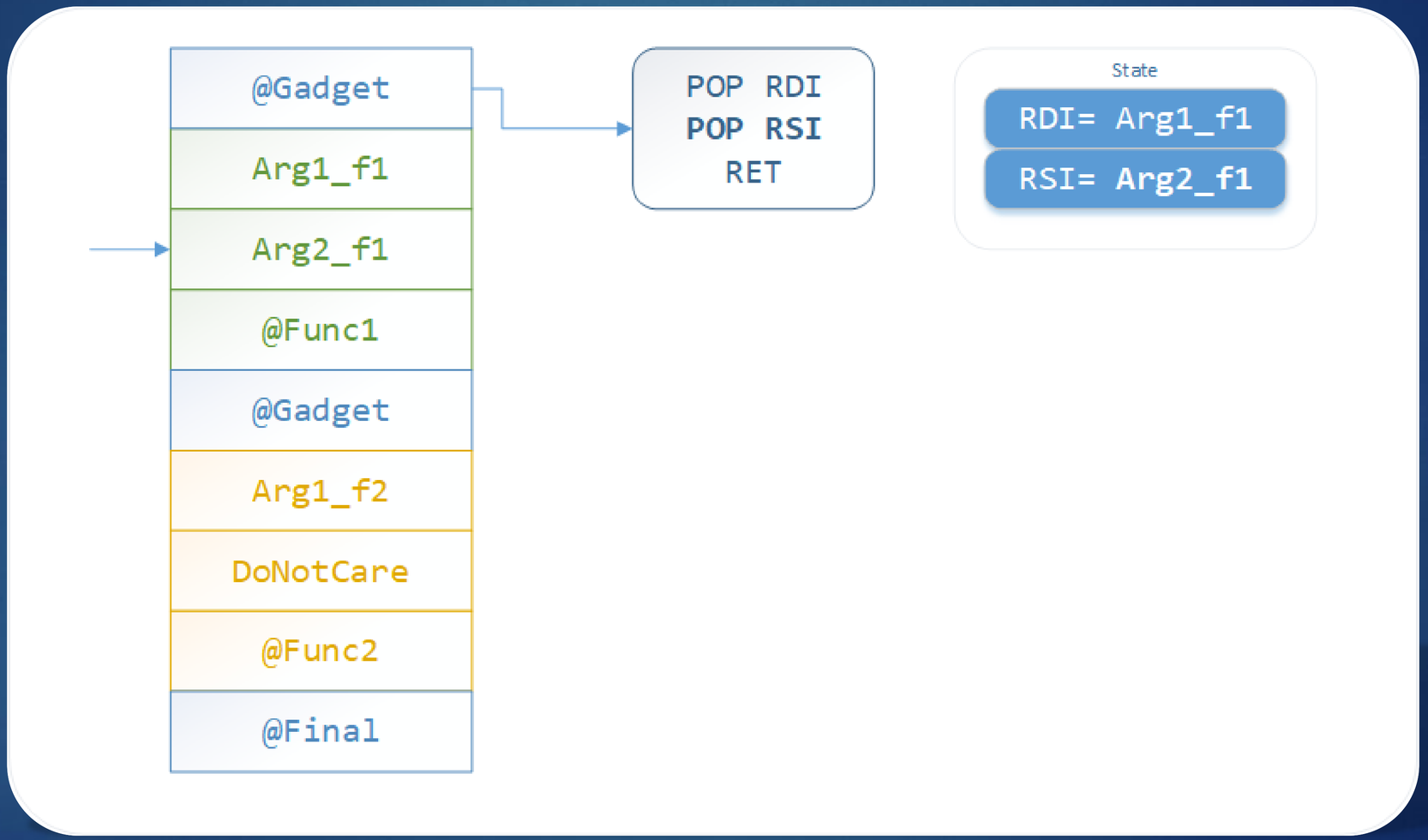
Basic gadget chains x64 (1/9)



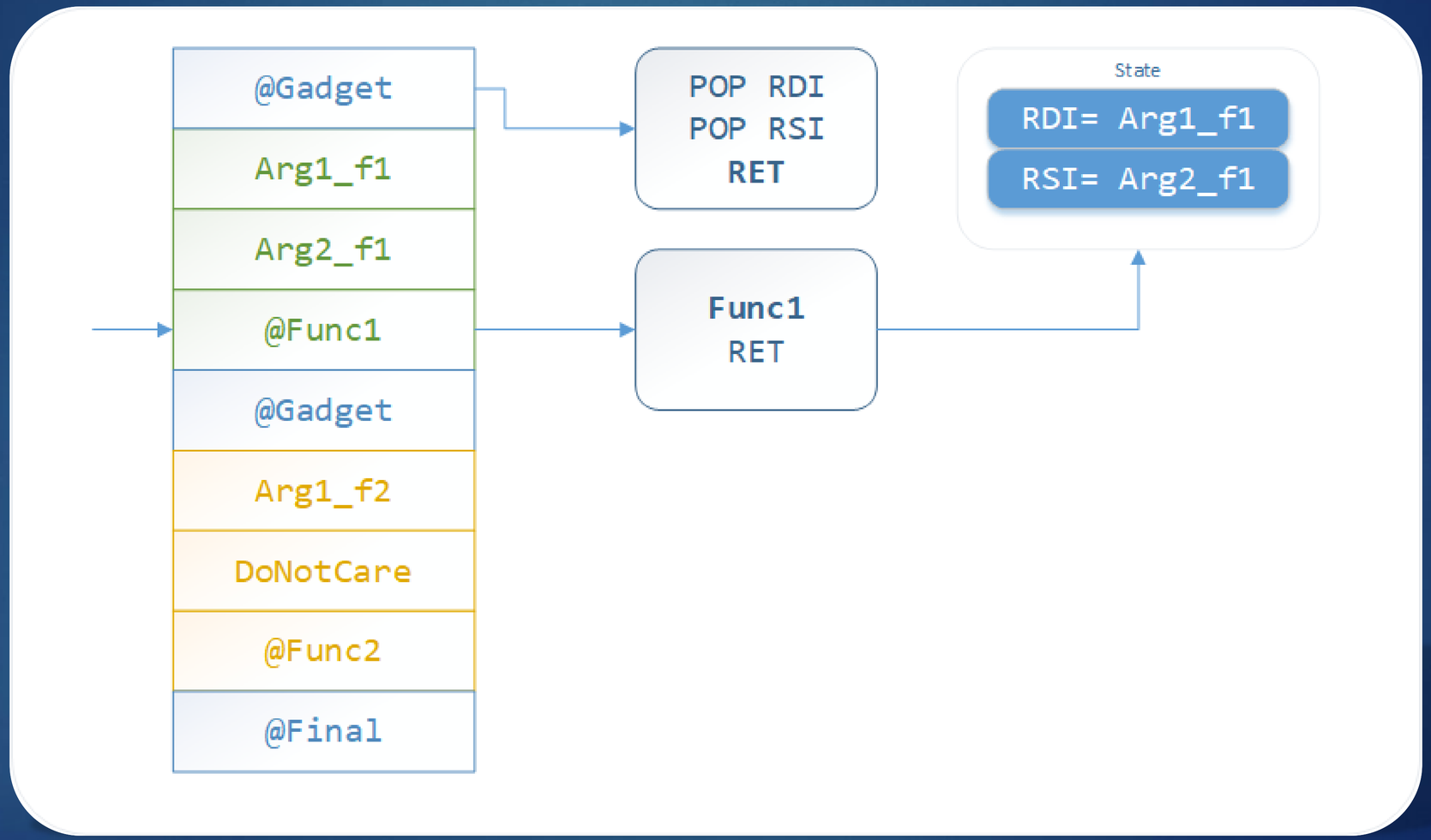
Basic gadget chains x64 (2/9)



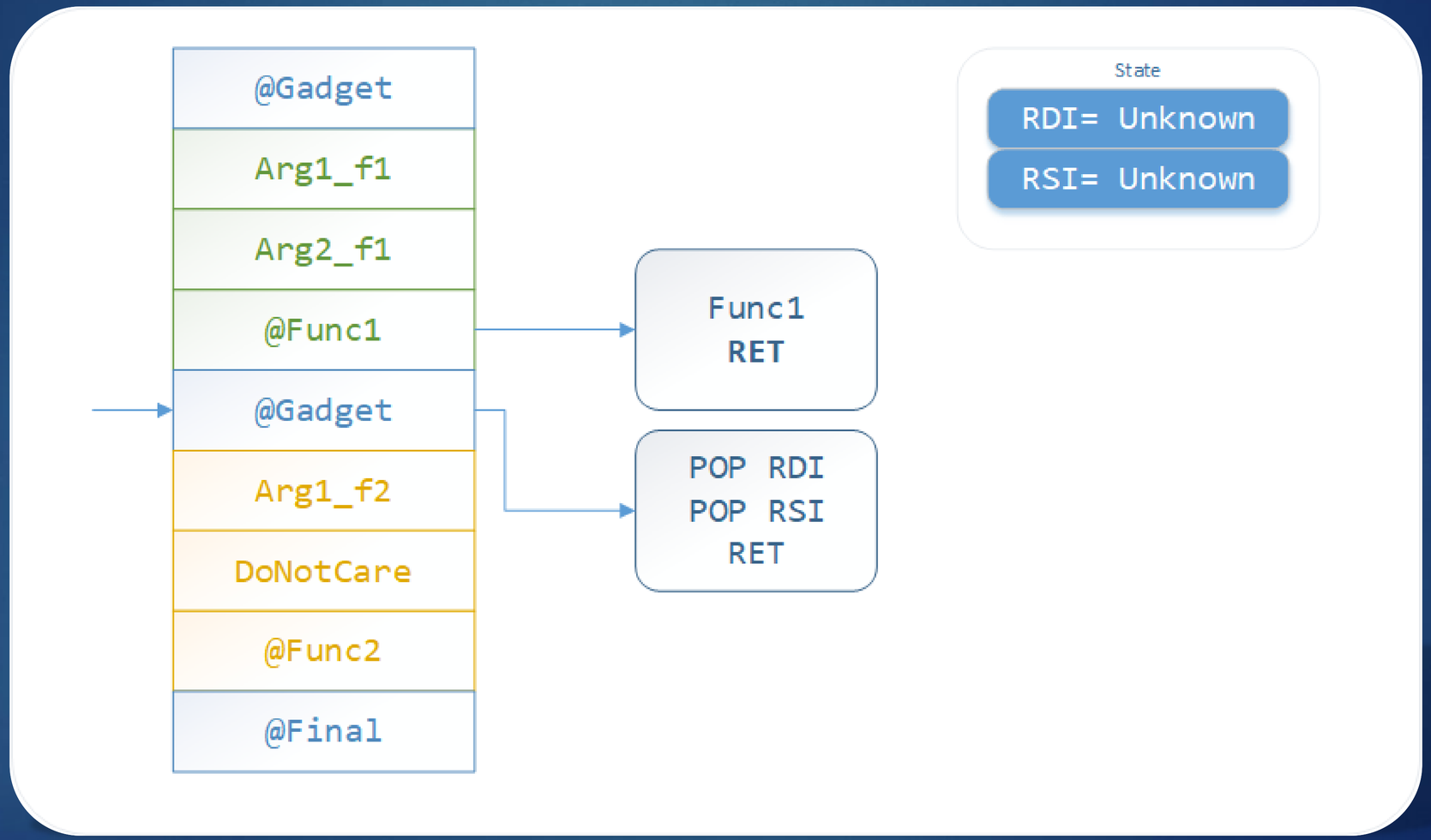
Basic gadget chains x64 (3/9)



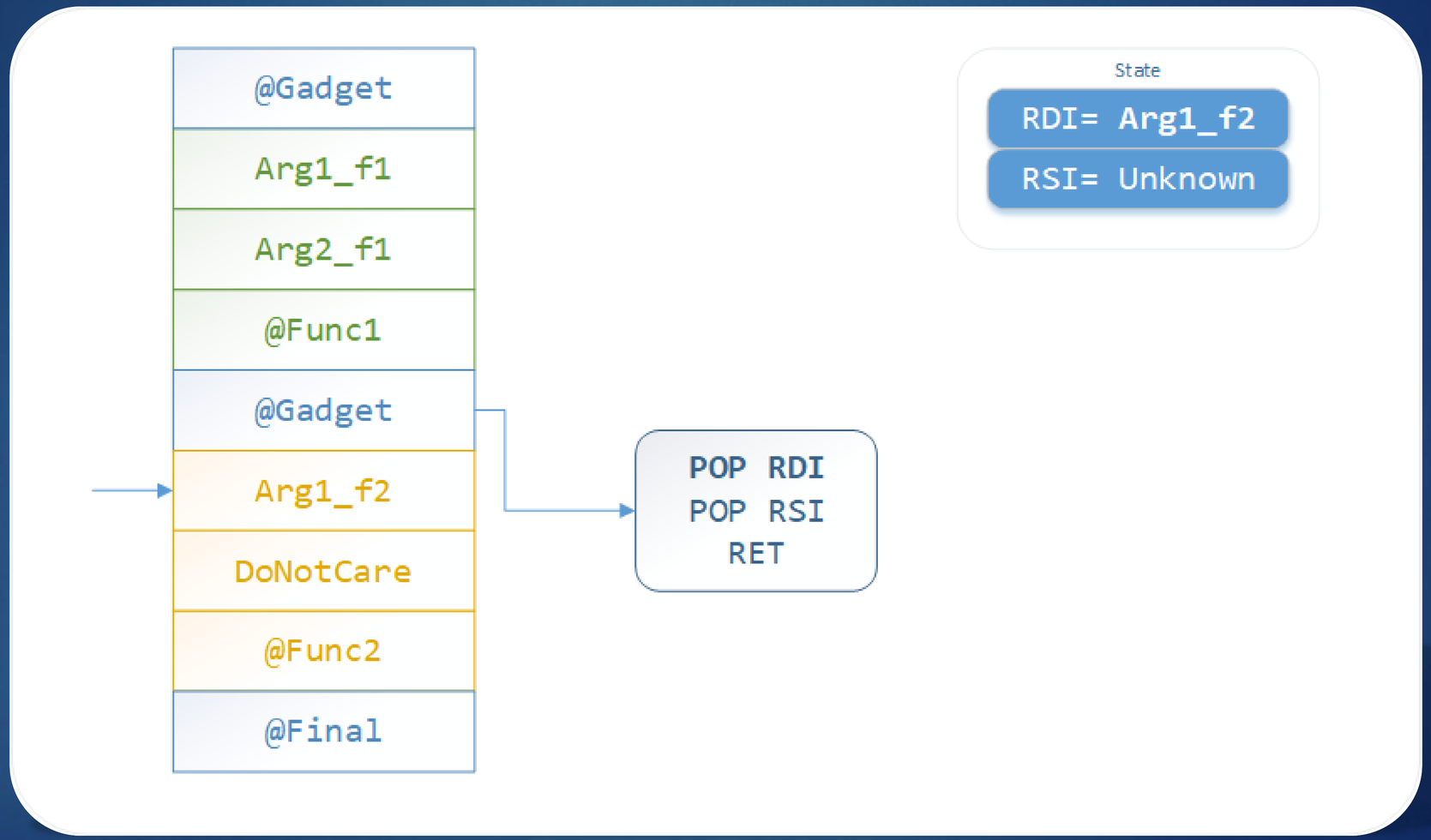
Basic gadget chains x64 (4/9)



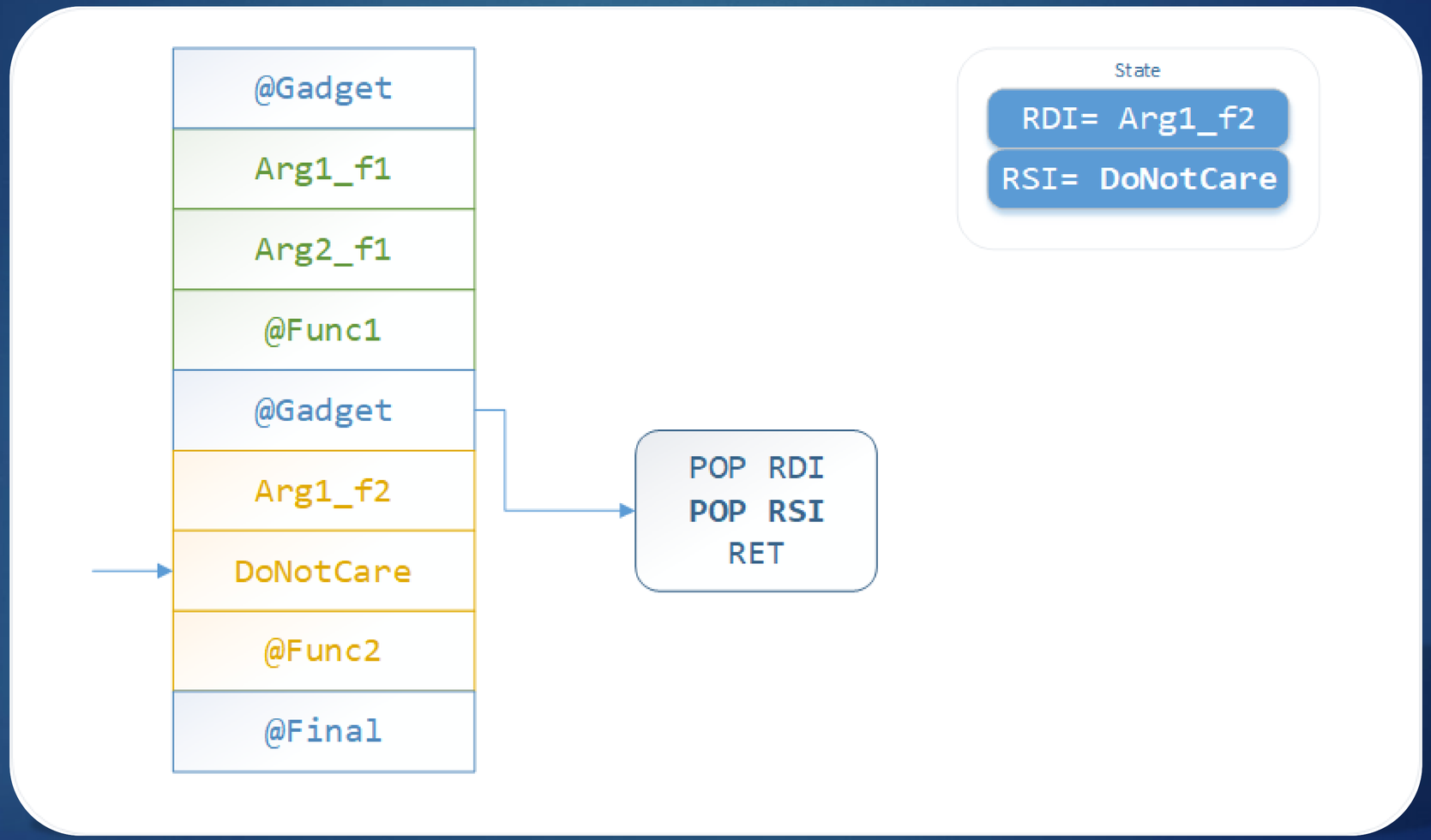
Basic gadget chains x64 (5/9)



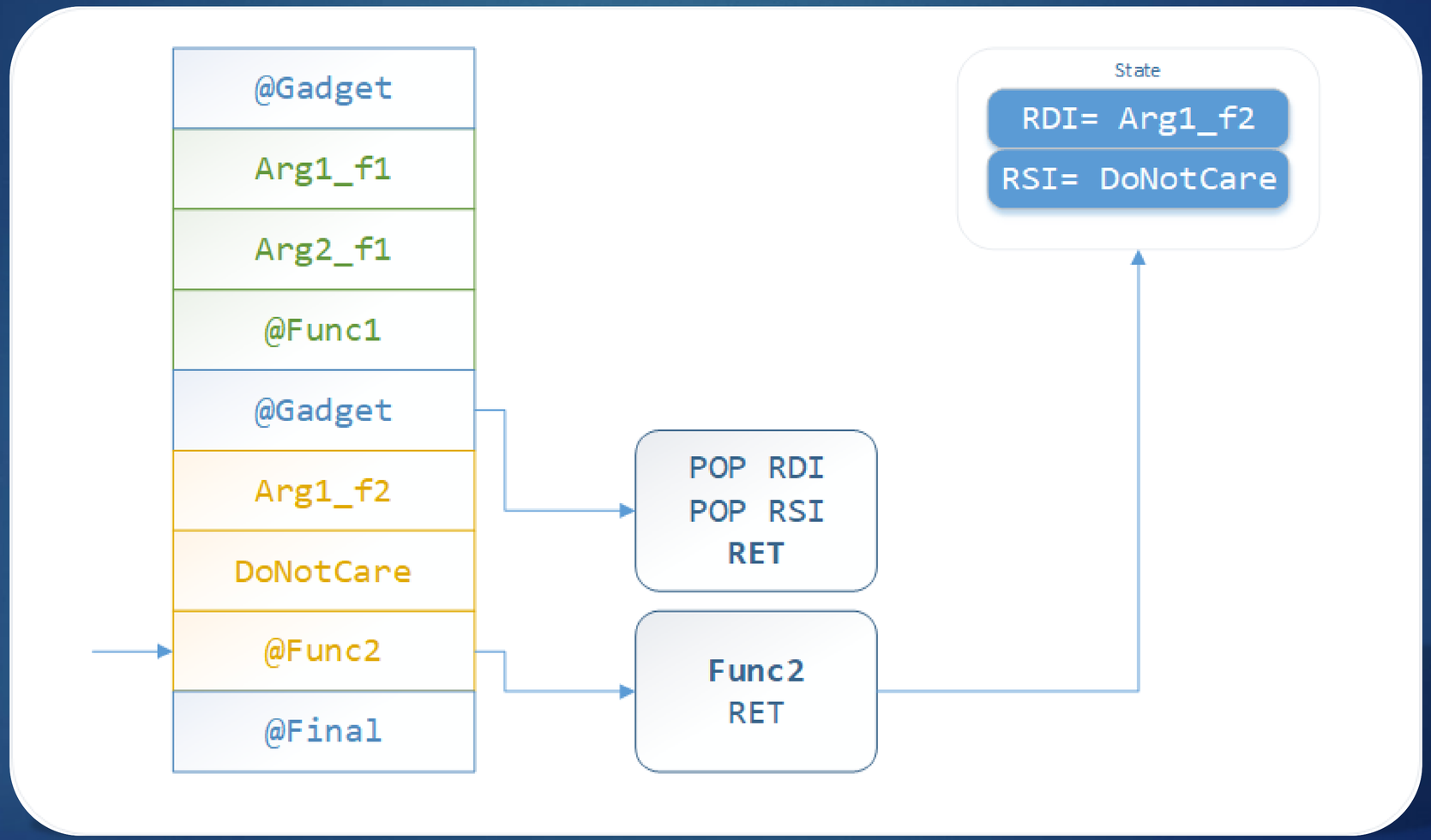
Basic gadget chains x64 (6/9)



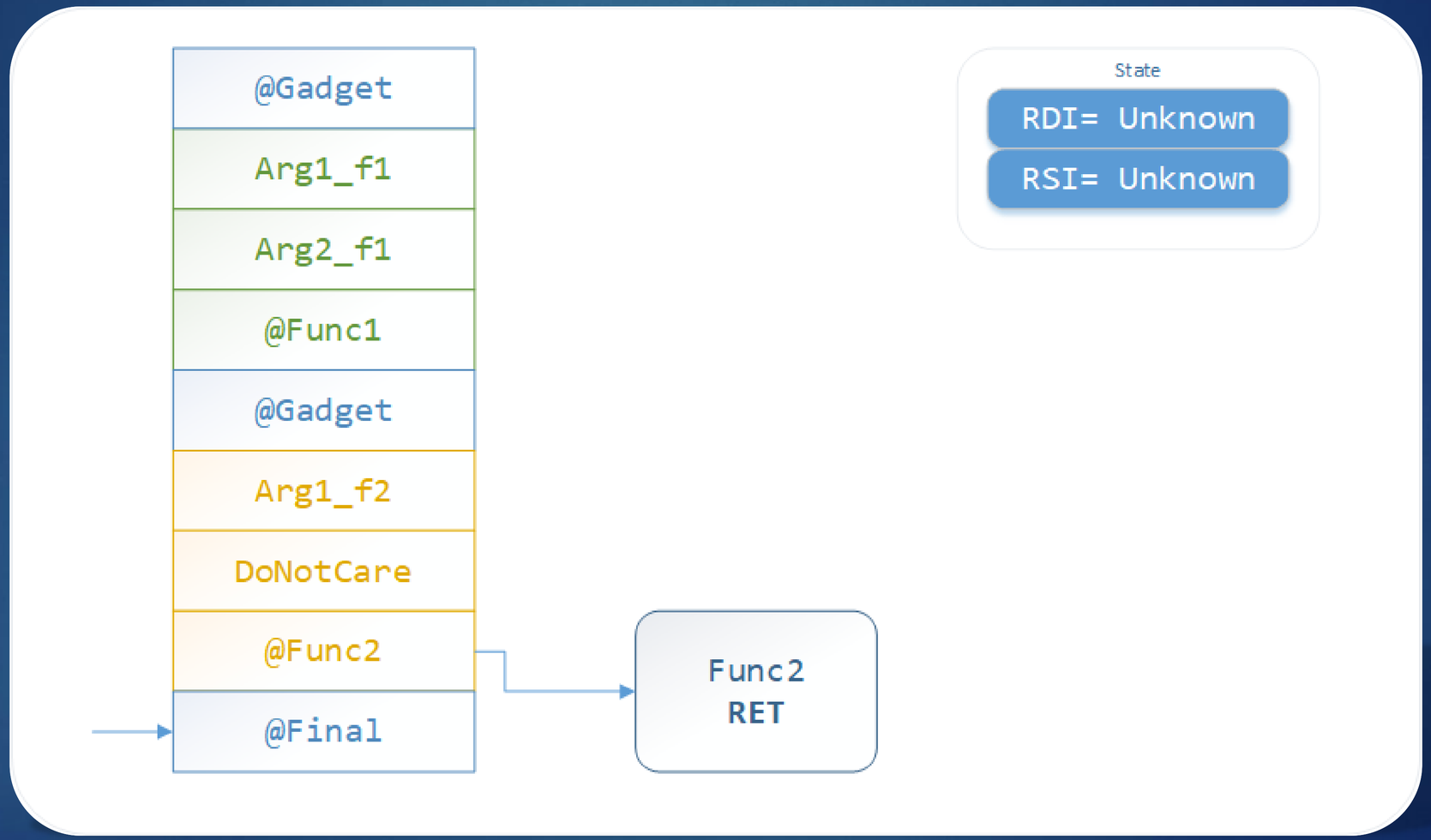
Basic gadget chains x64 (7/9)



Basic gadget chains x64 (8/9)



Basic gadget chains x64 (9/9)



Advice

Draw your stack/memory!

Debug your exploits.

Take the time to understand.

Practice!

Limitation of ASLR

Exercise on Windows:

1. Launch a *cmd.exe* process.
2. Using Process Hacker (https://processhacker.sourceforge.io/) look at the address of your *cmd.exe*.
3. Compare it to the address of *explorer.exe*.
4. Now compare the addresses of their *kernel32.dll* library.

- What happens here **?**
- What are the advantages for an attacker **?**

Practice!

ROP

- 1 exercises with 3 levels:
  - rop100, rop200, rop300
- Modified exercise from DefCon 2013.
- Goal: call *system* from the libc.
- Probably an infinite number of solutions.

Practice Debrief

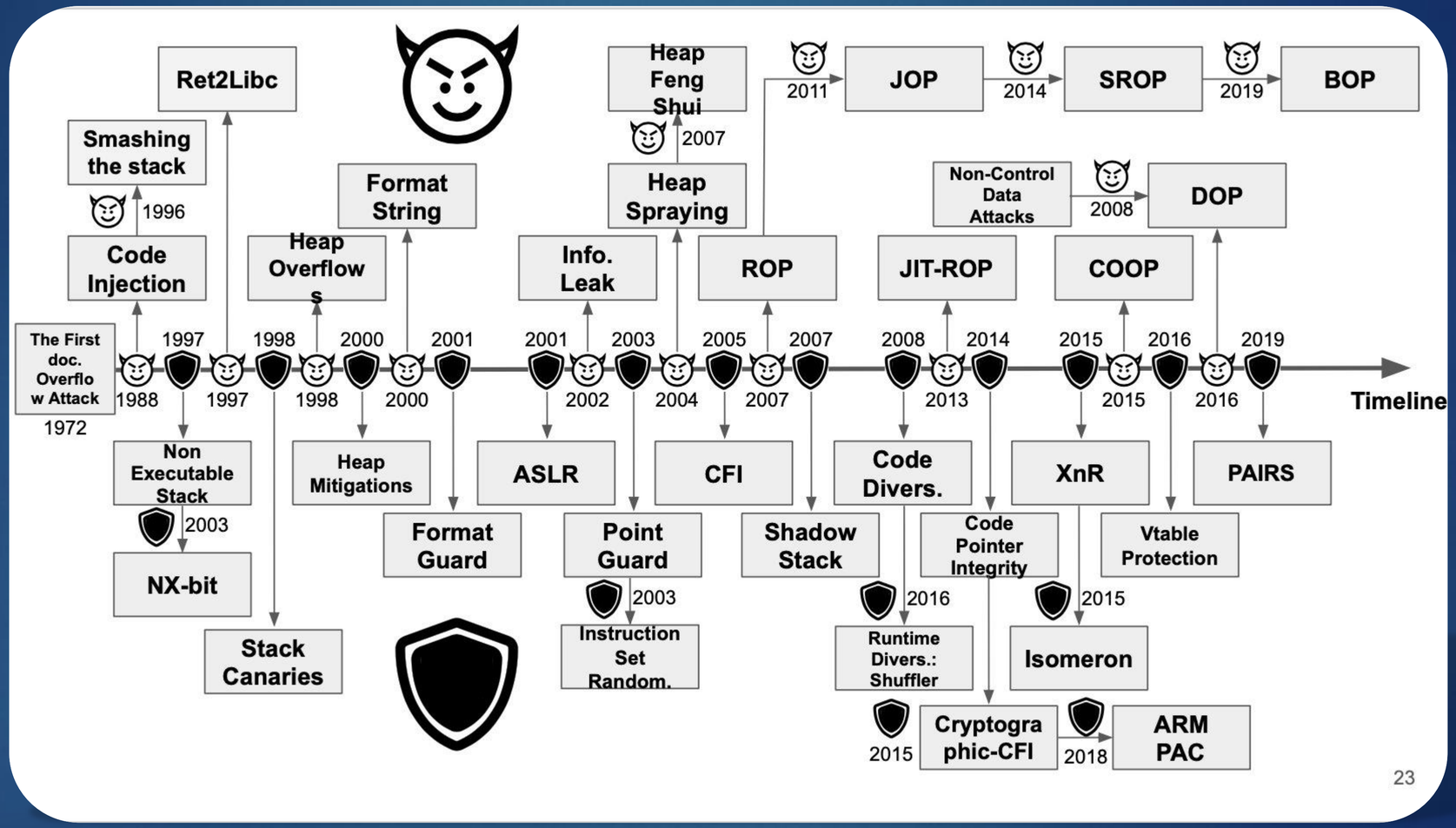
Going further: How does the linking on Windows works **?**

Could we do instruction splitting on ARM **?** Is there any other techniques we could use **?**

Plan

01	02	03	04	05	06
ASM Reminder	Tools	Shellcode	Basic Stack Buffer Overflow	Format String	ASLR, PIE & ROP

Evolution of protection



By Mohamed Hassan (@M_TarekIbnZiad)

Conclusion

- Stack buffer overflows are the most basic memory corruption vulnerabilities.
  - Still discovered today.
  - But not the most common anymore…
- Mitigations have made exploitations more complicated.
  - Sometimes impossible.
  - Often more specific conditions or added requirements.
- A good understanding is MANDATORY:
  - of your system, architecture, …
  - about the protections and how they work.

Practice!

FreeFloatFTP Vulnerability

Information:

- Real world vulnerability.
- This is a stack buffer overflow in a FTP server.
- Download FTPServer.exe.

Questions/Tasks:

- Where is the vulnerability **?**
  - How are you going to find it **?**
  - Which mitigations are enabled **?**
- Trigger a crash.
- Write an exploit as if you where in remote.
  - Goal is to launch an arbitrary program in the context of the server.

THANK YOU



Bruno Pujos