PwnSch00l

https://sigint.mx/pwn/

Binary Hacking 0x01

Past challenge walkthrough
Modern security mitigations (and how to defeat them)

DISCLAIMER

- All of the material in this school can be used for good (testing, research, educating), but also for bad
- We use our skills and knowledge responsibly and ethically
- We recommend you do the same
- We are not responsible for anything you do as a result of these lessons

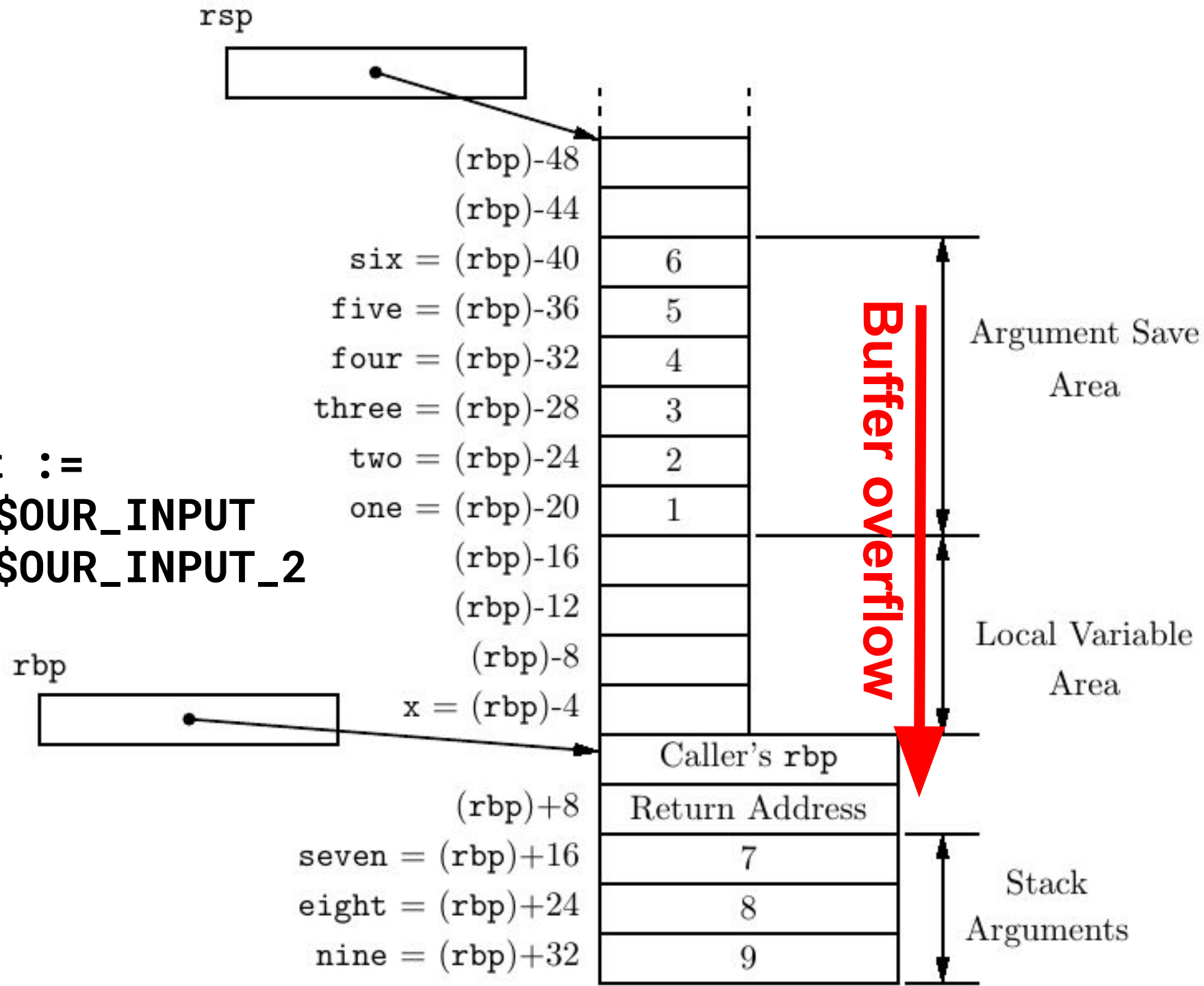
Words in *italic* are terms to search for

- Like last time, this lesson is part lecture, part workshop
- Ask questions
- Can you hear me at the back?

Last on PwnSch00l

- Reverse Engineering - understanding what a program does
- Pwn - making the program do what we want
- Static RE tools: **Radare2 + Cutter**, Binary Ninja, **IDA Pro**, RetDec
- Dynamic RE tools: **GDB + gef**, x64dbg, WinDbg, lldb
- Pwn tools: **python2-pwntools**
- Using stack overflow to overwrite return address

```
leave; ret :=
   RBP <- $OUR_INPUT
   RIP <- $OUR_INPUT_2
```



Today

- Stack protections:
    - Stack canaries, NX/DEP
- Past challenge walkthrough
- Virtual memory and paging
    - Protections: ASLR, PIE/PIC
- More stack overflow!
- Signed integer & overflow bugs

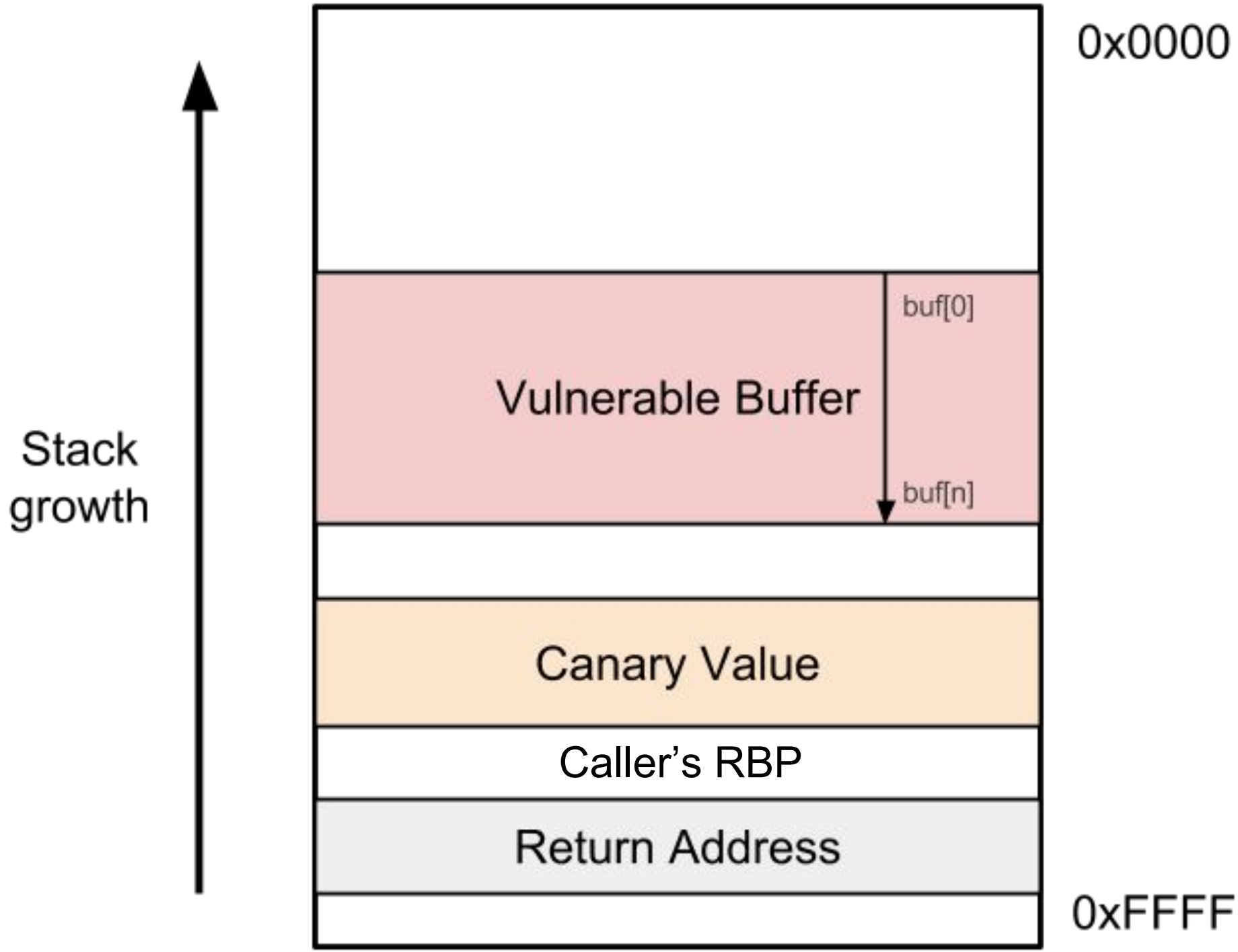
Stack Canaries

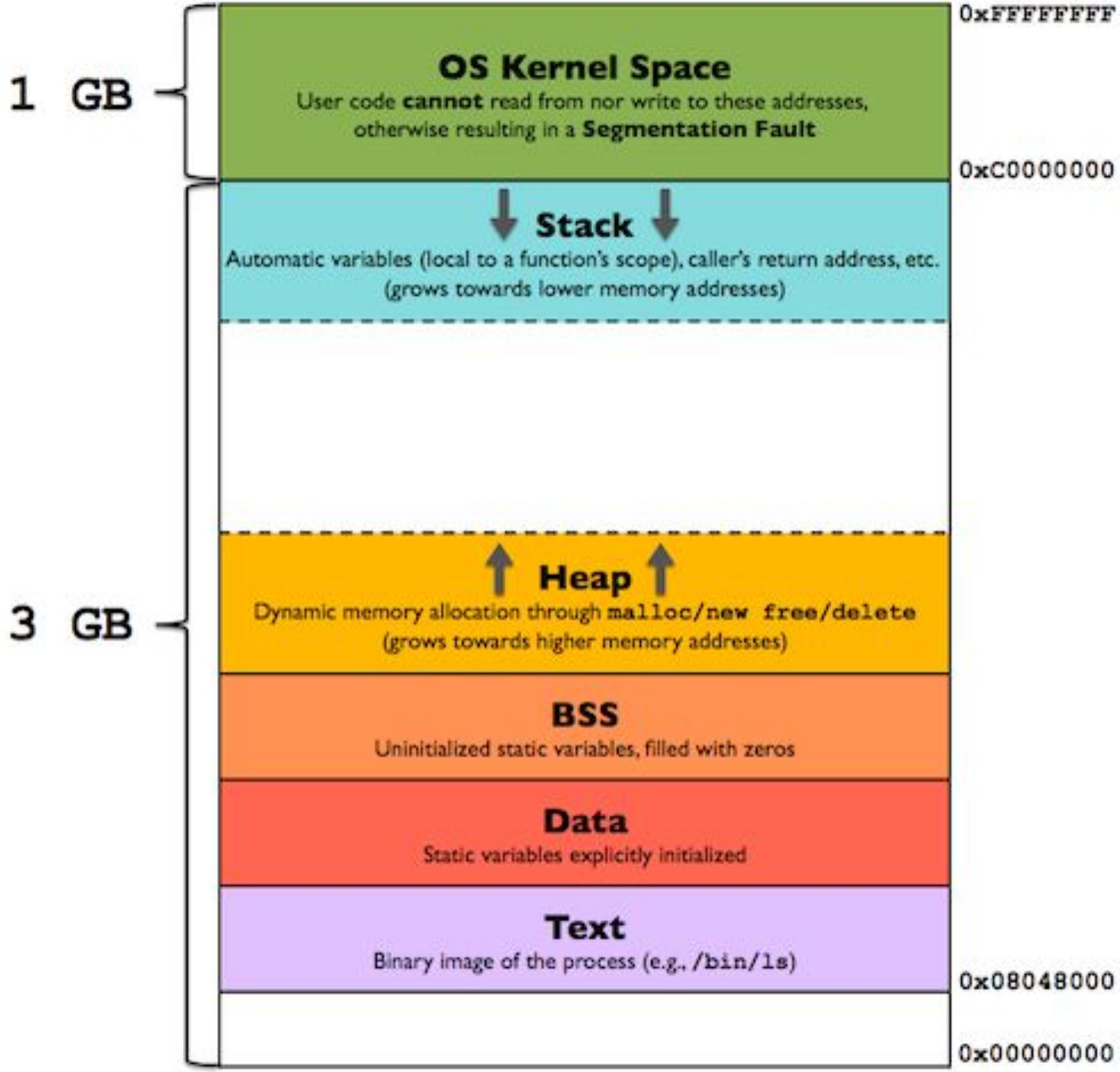
A secret (random - different every time the program runs) value on the stack that protects the stored return address and caller's RBP. Before the function returns, it is checked. If it was modified, the program crashes.

Defeated via:

- stack leak
- write-what-where (arbitrary write)

Try it yourself in the **`bird_store`** challenge.


0x0000
Stack
growth
buf[0]
Vulnerable Buffer
buf[n]
Canary Value
Caller’s RBP
Return Address
0xFFFF


1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

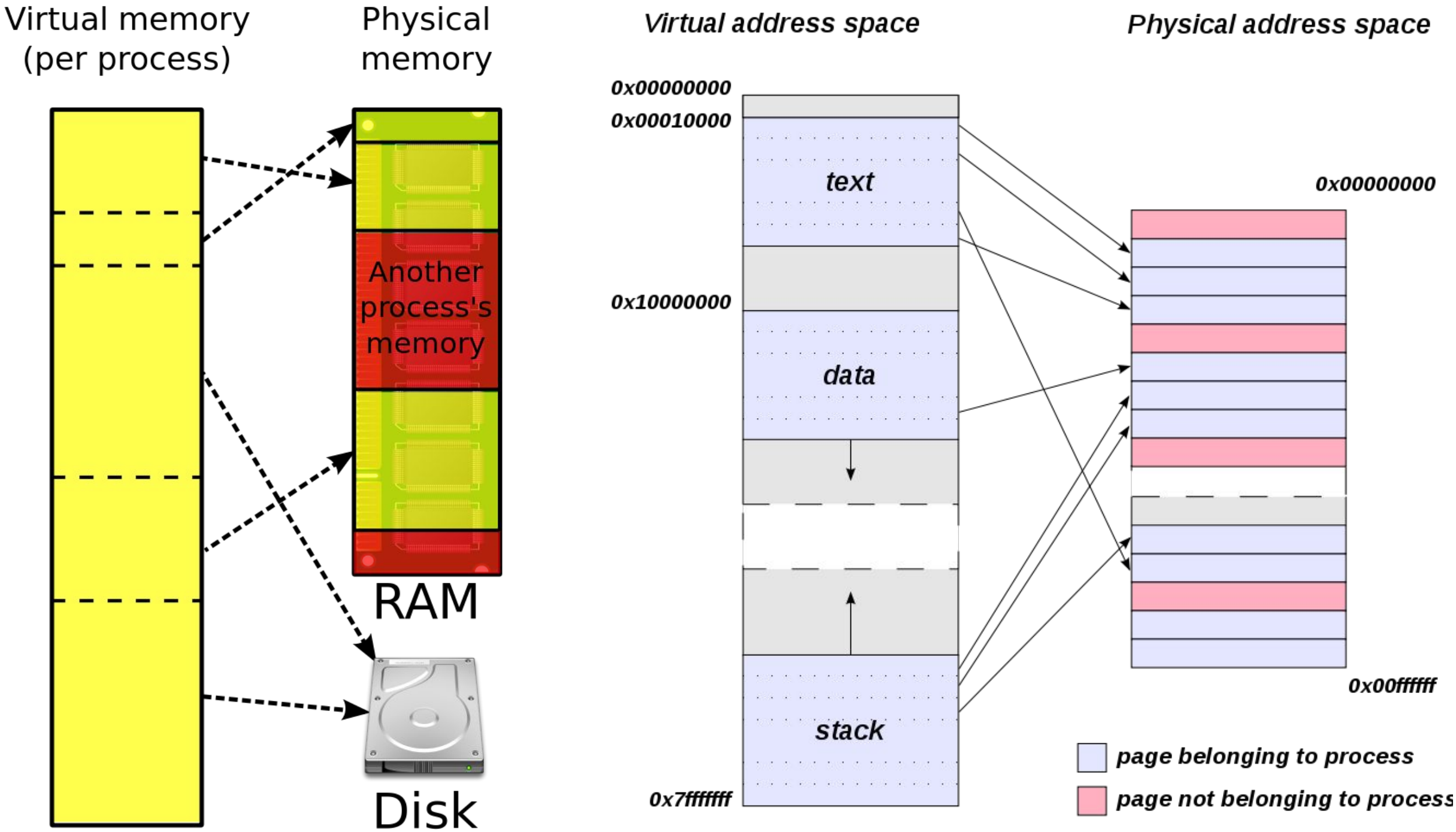
Paging

A way to:

- Separate processes (don’t want Chrome to access Excel’s memory)
- Implement an illusion of having more memory than there actually is

Each process (i.e. running program) gets its own ***virtual memory space***. A virtual address in this space corresponds to some address in **physical memory**, but physical addresses are unknown to the program (only the OS knows them).

Memory is mapped in units of 4 KiB (usually) called **pages**.


Virtual memory
(per process)
Physical
memory
Another
process's
memory
RAM
Disk
Virtual address space
Physical address space
0x00000000
0x00010000
0x10000000
0x7fffffff
text
data
stack
0x00000000
0x00ffffff
page belonging to process
page not belonging to process

ASLR + PIE/PIC

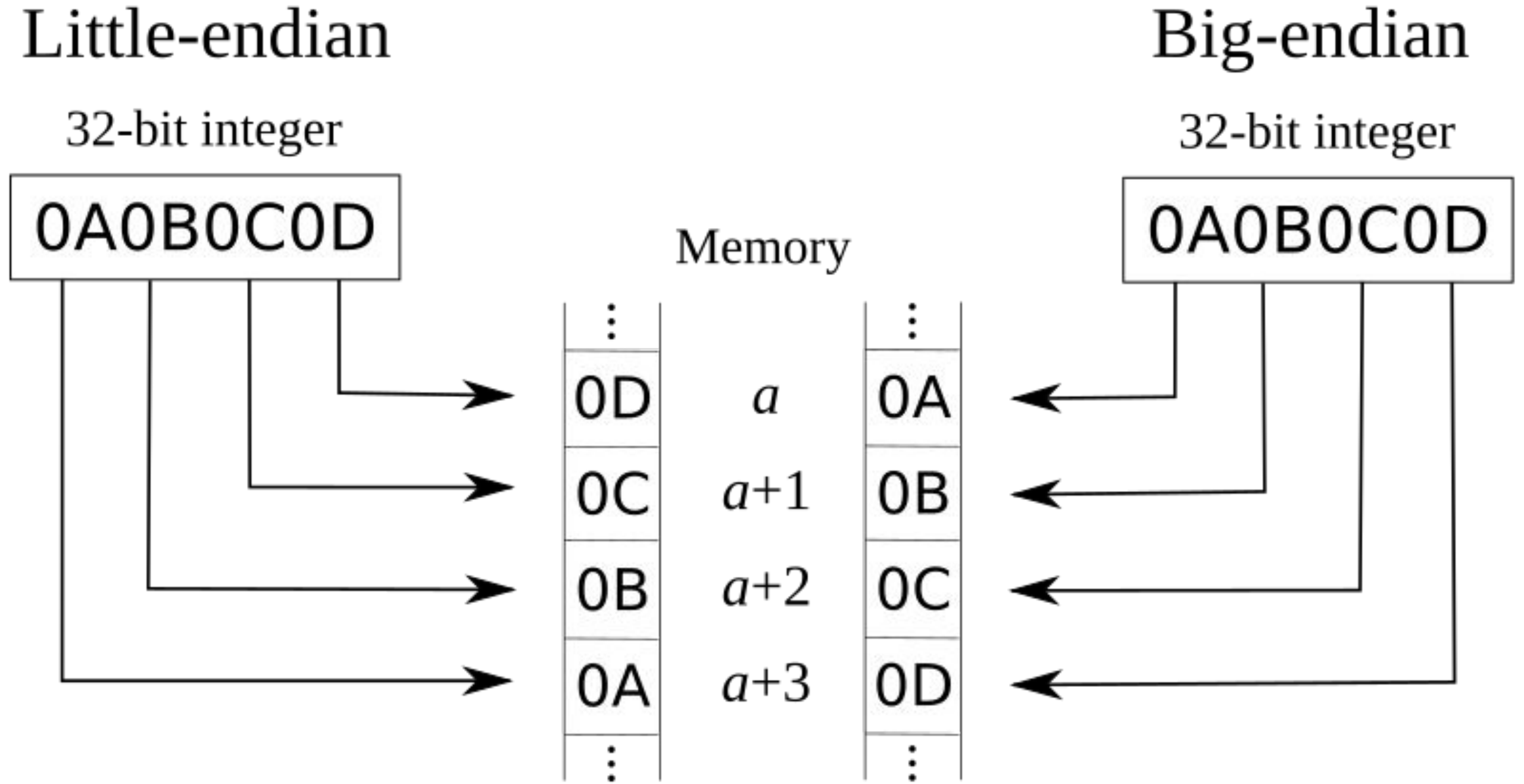
Address **S**pace **L**ayout **R**andomization - pages are placed at random addresses. As the attacker, we don't know where things are, so can't send malicious pointers to anything.

Position **I**ndependent **E**xecutable (or **C**ode) - a program that uses ASLR.

Defeated via:

- Leaks
- Partial overwrites
- Utilising crash state

Try partial overwrite in the **`notepad`** challenge.


Little-endian
Big-endian
32-bit integer
32-bit integer
0A0B0C0D
0A0B0C0D
Memory
0D
0C
0B
0A
a
a+1
a+2
a+3
0A
0B
0C
0D

NX/DEP

Non-e**X**ecutable bit (Linux) aka **D**ata **E**xecution **P**revention (Windows)

Marks sections as only containing data, but not code. Setting RIP to an address in any of these sections immediately crashes - can’t execute.

Defeats: shellcoding

Defeated via: ROP, JOP

On x86_64 implemented in page table, on x86 as a kernel hack (https://pax.grsecurity.net/docs/pageexec.txt).

Which mitigations enabled?

```
vagrant@vagrant:~$ checksec notepad
[*] '/home/vagrant/notepad'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: PIE enabled
```

The **checksec** tool is in pwntools, GEF and can also be downloaded separately.

Signed integer errors

What should have been an unsigned integer is actually signed, or vice-versa. Alternatively, a signed integer that should be positive is not checked for being positive.

Used to appear often, not so much nowadays thanks to improved compiler checks, fuzzing, etc.

Sometimes allow bypassing buffer length checks.

Useful for the **`notepad`** challenge.

File **test.c**

```c
char SECRET_BUF[64] = "SUPER_SECRET_P455W0RD...";
char BUF[64] = "PUBLIC_DATA_EVERYONE_CAN_READ...";

int idx = read_int_from_user(); // signed 32-bit integer
                                // user sends -8 (negative 8)

if (idx > 63) {                 // check passes
    puts("Hackers detected. Quite!");
    exit(0);
}

puts(&BUF[idx]);                // potentially sensitive data is leaked
                                // (depends on stack layout - not guaranteed in this case)
```

https://school.sigint.mx - our challenges (two for today, **notepad** will be enabled in a few minutes)

https://www.radare.org/r/ - Radare 2 | https://github.com/radareorg/cutter - Cutter GUI

https://www.hex-rays.com/products/ida/support/download.shtml - IDA Free

https://github.com/Gallopsled/pwntools - Python 2 library: pwntools

https://github.com/hugsy/gef - GDB overlay: gef <- makes GDB much nicer

https://github.com/RPISEC/MBE - Modern Binary Exploitation course | https://ctftime.org - CTFs and writeups

https://www.felixcloutier.com/x86/ - good X86 reference

Online challenges to practice on:

http://pwnable.tw | http://pwnable.kr | https://ropemporium.com/index.html

https://github.com/scwuaptx/HITCON-Training | *crackme* <- search this for reversing challenges

If you would like to play some CTFs with us, join the Slack:
https://siginthqv2.slack.com or ask on social media.