Introduction to binary exploitation on Linux

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

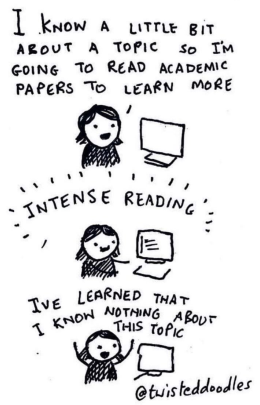
Outline

1. Memory corruption attacks
   - Format-string attacks

2. Mitigations
   - Stack canaries
   - Non-executable stack
   - Address-Space Layout Randomization
   - Control-flow Integrity

3. Learning resources

Memory corruption attacks is a *huge* topic



https://twitter.com/ap_taber/status/1183138126593552384

See, for instance, [SPWS13] for a survey of memory corruption attacks

Recap

So far, we saw how a (stack-based) BOF may allow an attacker to

- change the value of (local) variables
- hijack the control-flow of the program, to execute
  - existing code
  - new, injected, code

What about

- global variables?
- heap-allocated objects?

Buffer overflows

- BOFs can be everywhere: stack, data and heap

Heap exploitation

heap exploitation usually means exploiting the memory allocator, by corrupting/leaking its metadata, and that is a more advanced topic

- pointers may point to different regions
  - don't forget function-pointers (GOT, malloc hooks, v-tables, ...)
- Use-After-Free (UAF) bugs can make different objects overlap

ASLR

ASLR is a security mitigation that randomizes addresses; *however*, if the program is non-PIE, you can

- ignore ASLR for code/data regions
  - `checksec` (pwntools/GEF) is your friend
- call libc functions used by the program, by using PLT addresses

With pwntools, given an `ELF` object `e`, you can use

- `e.plt.`*`name`* and `e.got.`*`name`*, for PLT and GOT entries
- if the binary is not stripped, also `e.sym.`*`name`* for everything

to find the addresses by name

PIE

If the binary is PIE, you get the offsets from the base address instead

Example

```c
#include <stdio.h>

char other_secret[] = "This secret is not referenced on the stack...\n";

int main()
{
	char *s_str = "This is a secret string! (pointer on the stack)\n";
	long s_int = 0xc0ffee;
	char name[16];
	printf("What's your name? ");
	fflush(stdout);
	fgets(name, sizeof(name), stdin);
	printf("Hi, ");
	printf(name);
}
```

Do you see any bugs? Can we obtain the values of `s_int`, `s_str` and `other_secret` by simply running the program?

Format strings

From `printf(3)`:

- . . . format string consists 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. hn → short, `hhn` → char)
  - 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

glibc-specific (more a bug than a feature): %**blank* seems to work as %%%d*blank* i.e., it prints a %, an integer and the blank

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`
    - or even `0xde`, `0xad`, `0xbe` and `0xef`
  - format string is on the stack
    - its content is under our control
    - can be reached by some arguments

See [New00] and
`https://docs.pwntools.com/en/stable/fmtstr.html`

`printf` is (almost) Turing complete!

See "Control-flow bending" [CBP+15], and a tic-tac-toe implemented by a *single* call to `printf`: `https://github.com/carlini/printf-tac-toe`

Examples/Exercises

→ `fmt-strings/example.c`

- `find_int_secret.py`
- `find_str_secret.py`
- `find_buf_offset.py`
- `print_other_string.py`

→ `printfun`, from TUCTF 2019

Outline

1. Memory corruption attacks
   - Format-string attacks
2. Mitigations
   - Stack canaries
   - Non-executable stack
   - Address-Space Layout Randomization
   - Control-flow Integrity
3. Learning resources

_FORTIFY_SOURCE

See `feature_test_macros(7)`:

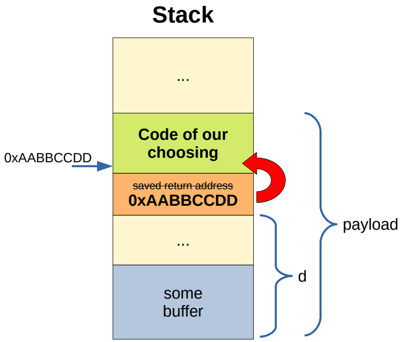
> *_FORTIFY_SOURCE (since glibc 2.3.4)*
> *Defining this macro causes some lightweight checks to be performed to detect some buffer overflow errors when employing various string and memory manipulation functions ...*
> *Some of the checks can be performed at compile time (via macros logic implemented in header files), and result in compiler warnings; other checks take place at run time, and result in a run-time error if the check fails.*

From `gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html`:

> *... with -D_FORTIFY_SOURCE=2, %n in format strings of the most common *printf family functions is allowed only if it is stored in read-only memory*

In Ubuntu $\geq$ 8.10 set by default, and activated when `-O` is set ≥ 2. To disable, either `-U_FORTIFY_SOURCE` or `-D_FORTIFY_SOURCE=0`

Let's recap a classic BOF attack



The "ingredients" for succeeding in a classic BOF attacks are:

1. an undetected overflow → stack canaries/cookies
2. an executable stack → NX/DEP/W^X
3. a predictable address → ASLR

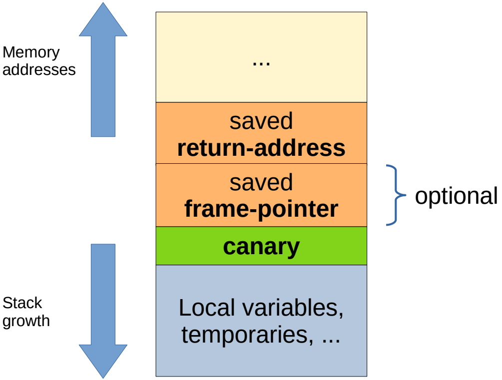
Stack canaries

- 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 “something” before the saved return pointer
- “something” is typically an integer containing
  - `'\0'` (in the first byte)
  - maybe bytes `'\x0a'`, `'\x0d'`
  - some random bytes
- in gcc see options `*stack-protector*`
  - In Ubuntu $\geq$ 14.10, `-fstack-protector-strong` is enabled by default

checksec

You can see what protection/mitigation mechanisms are enabled on an executable by using `checksec`

Visually


Memory
addresses
...
saved
return-address
saved
frame-pointer
optional
canary
Stack
growth
Local variables,
temporaries, ...

In code...

```
99: sym.get_name ();
        ; var int32_t var_4ch @ ebp-0x4c
        ; var int32_t canary @ ebp-0xc
        0x08048566      55             push ebp
        0x08048567      89e5           mov ebp, esp
        0x08048569      83ec58         sub esp, 0x58
        0x0804856c      65a114000000   mov eax, dword gs:[0x14]
        0x08048572      8945f4         mov dword [canary], eax
        0x08048575      31c0           xor eax, eax
        0x08048577      83ec0c         sub esp, 0xc
        0x0804857a      68b0860408     push str.Enter_your_name:   ; 0x80486b0 ; "Enter your name: "
        0x0804857f      e84cfeffff     call sym.imp.printf         ; int printf(const char *format)
        0x08048584      83c410         add esp, 0x10
        0x08048587      a130a00408     mov eax, dword [obj.stdout] ; obj.__TMC_END
                                                                   ; [0x804a030:4]=0
        0x0804858c      83ec0c         sub esp, 0xc
        0x0804858f      50             push eax
        0x08048590      e84bfeffff     call sym.imp.fflush         ; int fflush(FILE *stream)
        0x08048595      83c410         add esp, 0x10
        0x08048598      83ec0c         sub esp, 0xc
        0x0804859b      8d45b4         lea eax, [var_4ch]
        0x0804859e      50             push eax
        0x0804859f      e84cfeffff     call sym.imp.gets           ; char *gets(char *s)
        0x080485a4      83c410         add esp, 0x10
        0x080485a7      83ec0c         sub esp, 0xc
        0x080485aa      8d45b4         lea eax, [var_4ch]
        0x080485ad      50             push eax
        0x080485ae      e85dfeffff     call sym.imp.strdup         ; char *strdup(const char *src)
        0x080485b3      83c410         add esp, 0x10
        0x080485b6      8b55f4         mov edx, dword [canary]
        0x080485b9      653315140000.  xor edx, dword gs:[0x14]
     ┌─< 0x080485c0      7405           je 0x80485c7
     │   0x080485c2      e859feffff     call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)
     └─> 0x080485c7      c9             leave
        0x080485c8      c3             ret
```

What's `gs:[0x14]`?

gs refers to the Thread Control Block (TCB) header in 32 bit executables (`fs` in 64 bit ones)

```
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: `https://chao-tic.github.io/blog/2018/12/25/tls`

Tackling stack-canaries

Not sure-fire ways; yet, sometimes

- canaries can be
  - brute-forced
    - can be done one byte at a time on a `fork`ed server; a nice paper for hacking *blind* in these settings is [BBM+14]
  - obtained by exploiting leaks
- changing the value of a local variable is enough to alter execution flow
- indirect writes may allow to write *beyond* the canary

Moreover, not all buffers are on the stack ☺

More information

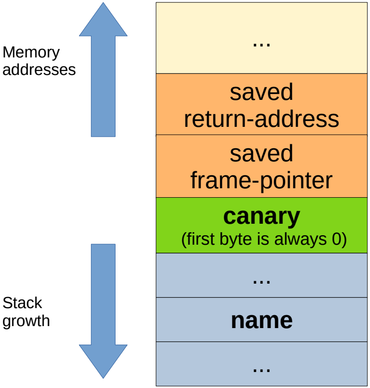
- To go deeper, in-depth explanation in Playing with canaries:
  `https://www.elttam.com/blog/playing-with-canaries/`
- For exploit developing/debugging
  - command `canary` in GEF
    `https://gef.readthedocs.io/en/master/commands/canary/`
  - from the above post/author:
    `https://github.com/elttam/canary-fun`; e.g.:
    `read_canary_from_pid.py`

Example: the best game v.0 (shortened for fitting the slide)

```c
#include ... /* 32 bits, Stack canary (no NX, no PIE) */

void spawn_shell() { /*...*/ }
void quiz() {
    char name[32];
    for (;;) {
        printf("Which is the best game?\nHZD\nTD2\nFIFA\n# ");
        memset(name, 0, sizeof(name));
        read(STDIN_FILENO, name, 320);
        if (strncmp(name, "FIFA", 4) == 0)
            printf("No way! TD2 >>> FIFA\n");
        else if (strncmp(name, "TD2", 3) == 0)
            printf("Are you kidding? HZD >>> TD2\n");
        else if (strncmp(name, "HZD", 3) == 0) {
            printf("I agree!\n");
            return;
        } else printf("LOL %sIs it even a game?!?\n", name);
    } }
```

TBG 0 (visually)



A template for *pwning*

```
#!/usr/bin/env python3
from pwn import *

EXE_FILENAME='...'
HOST = args.HOST or '...'
PORT = int(args.PORT or ...)
exe = context.binary = ELF(EXE_FILENAME)
argv = [EXE_FILENAME]
envp = {}
gdbscript = '''\
set startup-with-shell off
c
'''
def start():
    if args.GDB:
        return gdb.debug(args=argv, env=envp, gdbscript=gdbscript)
    if args.REMOTE:
        return remote(HOST, PORT)
    return process(argv=argv, env=envp)
io = start()
io.interactive()
```

Example: the best game v.0 PRO (shortened for fitting)

```
#include ... /* 32 bits, Stack canary (no NX, no PIE) */

// no more spawn_shell()

void quiz() { /*... as before ...*/ }
```

At least two ways to tackle the challenge:

Easier similar to what we saw in a previous lecture, leveraging the `call esp` at address `0x080488eb`

To search such instructions you can use:

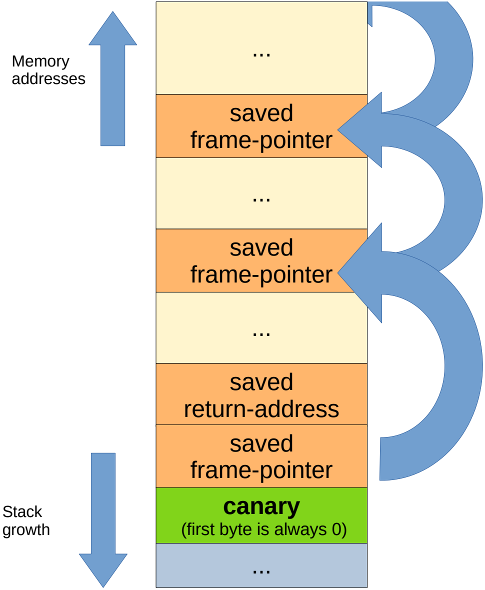
```
ropper -f best_game0_pro --type jop
          --search '%esp%' --quality 1
```

Interesting

```
b *(quiz+1)
run
telescope
```

- Hint: what will it be the value of esp after the return? Compare it with the saved (`main`'s) frame pointer out ☺

TBG 0 – PRO (visually)



Code reuse attacks (on NX stacks)

NX avoids to execute new code; see `ld` option `execstack`

It can be bypassed by reusing code:

- return to "something useful"
- return-to-libc
  - with ASLR enabled, finding the libc in memory can be tricky
  - PLT entries are at fixed addresses, in non-PIE binaries
- ROP: Return Oriented Programming [Sha07]
  To learn/practice ROP: `https://ropemporium.com/`
  - You may also want to checkout *Sigreturn Oriented Programming (SROP)* [BB14]

Existing code could be used to *remove* NX

E.g. see `mprotect(2)`

ROP

Idea: chaining gadgets to create “new” code [Sha07]

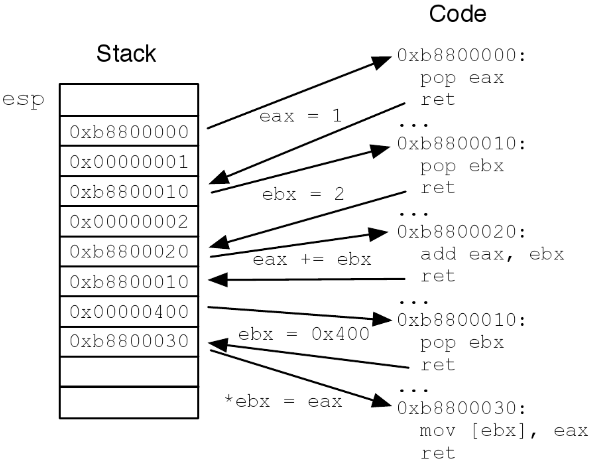


Image taken from [Pap15]

Ropper

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

E.g.:

```
ropper --file best_game2 --search 'pop rdi'
ropper --file ...  --chain execve --badbytes 000a0d
```

Bad bytes

It seems `ropper` doesn't handle well the cases where the bad bytes are part of addresses.
Always double-check the output of your tools.

Pwntools

Unsurprisingly ☺ Pwntools can help with ROP too:
`https://docs.pwntools.com/en/stable/rop/rop.html`

Stack alignment on x86

ABIs requires 16-byte stack alignment:

- 32 bits

  > *The end of the input argument area shall be aligned on a 16 (32 or 64, if __m256 or __m512 is passed on stack) byte boundary. In other words, the value (%esp + 4) is always a multiple of 16 (32 or 64) when control is transferred to the function entry point.*

- 64 bits

  > *. . . In other words, the value (%rsp + 8) is always a multiple of 16 (32 or 64) when control is transferred to the function entry point.*

Local vs remote exploits

A misaligned stack is a possible reason that can make an exploit fail remotely, even if it is working locally. You may add "NOPs" (addresses of a `ret` instruction) to align the stack before entering a libc function

Example: the best game v.1 (shortened for fitting the slide)

```c
#include ... /* 32 bits, NX (no stack canary, no PIE) */

char no_more_spawn_shell_ahahah[] = "/bin/sh";

void quiz()
{
    char name[32];
    for (;;) {
        system("echo 'Which is the best game?\n...\n# '");
        memset(name, 0, sizeof(name));
        read(STDIN_FILENO, name, 320);
        if (strncmp(name, "FIFA", 4) == 0)
            printf("No way! TD2 >>> FIFA\n");
/* ... as before ... */
```

Example: the best game v.2

Same... 64 bits. Is it so different?

Address-Space Layout Randomization

> *In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries*
>
> `https://en.wikipedia.org/wiki/Address_space_layout_randomization`

See gcc options `*pie*`; `-pie` is default since Ubuntu 16.10 (?)

> *CTFs are the only place that I'm happy to not have PIE*
>
> `https://twitter.com/jf100w/status/1190741258148438016?s=03`

ASLR Attacks

- brute-forcing
  - may be possible on 32 bits, since entropy should be 19 bits; see, for instance, `https://stackoverflow.com/a/37512642`
  - n.a. on 64 bit executables
- partial overwriting
  - by overwriting the two least significant bytes (=16 bits), which come first in little-endian, you have a $\frac{1}{2^4}$ probability of bypassing ASLR
    - sometimes a byte is enough, and that's sure-fire
- leaking via `/proc/`*`pid`*`/...`
  - See, for instance, `https://blog.blazeinfosec.com/the-never-ending-problems-of-local-aslr-holes-in-linux/`
- two-stage attacks: leak + code reuse; idea:
  - leak the address of some prog/libc function
  - find the base of prog/libc for the process
  - calculate the address of `system`/*other-interesting-functions*
  - profit ☺
- exploit the dynamic loader: "leakless" [DFCS+15]

Example: the best game v.2 PIE

As v.2, but PIE

Example: the best game v.3 (shortened for fitting the slide)

```
#include ... /* 64 bits, NX, PIE (no Stack canary) */

void quiz()
{
    char name[32];
    for (;;) {
        printf("Which is the best game?\nHZD\nTD2\nFIFA\n# ");
        memset(name, 0, sizeof(name));
        /*... as before ...*/
    }
}
```

This time, there is no `system` in the PLT

Exploit development with different libc versions

Two main possibilities:

- Distinguishing the local (process/GDB) case, wrt the `REMOTE` one in the script; i.e. using different ELFs and offsets in your script
- Running the same version, different from yours
  1. find the pair loader+libc, for instance, by using *libc database* `https://github.com/niklasb/libc-database`
     - `./identify ...`
     - `./download ...`
     - `(./dump ...)`
  2. run the corresponding `ld-linux.so`, by using `patchelf(1)`, to change interpreter (`--set-interpreter`) *and* the library version (`--replace-needed`)

Don't trust the so-called `LD_PRELOAD` "trick"

Loading the libc, without using the corresponding loader, is a simple, yet *unreliable*, method since many libc versions depend-on their specific loader

One-gadget

When playing CTFs, an extremely handy gadget is the one-gadget RCE that calls `execve('/bin/sh', NULL, NULL)`
The following tool provides such gadget finder for x86/64:

https://github.com/david942j/one_gadget

Usage example

```
$ one_gadget libc.so.6
...
0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL
...
```

To install:

```
# export GEM_HOME=~/ruby-gems
# export PATH=$PATH:$GEM_HOME/bin
gem install one_gadget # requires ruby version >= 2.1.0
```

Example: the best game v.4 (shortened for fitting the slide)

```c
#include ... /* 64 bits, NX (no Stack canary, no PIE) */

void quiz()
{
    char name[32];
    printf("Which is the best game?\nHZD\nTD2\nFIFA\n# ");
    memset(name, 0, sizeof(name));
    read(STDIN_FILENO, name, 320);
        /*... as before ...*/
}
```

CFI

> *CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time. The CFG in question can be defined by analysis — source code analysis, binary analysis, or execution profiling. [ABEL05]*

Nice idea, however [CBP$^+$15] shows that

> *an attacker can leverage a memory corruption vulnerability to achieve Turing-complete computation on memory using just calls to the standard library . . .*
>
> *shadow stacks in combination with CFI and find that their presence for security is necessary: deploying shadow stacks removes arbitrary code execution capabilities of attackers in three of six cases*

If you're curious about `printf` being Turing-complete, it's a must-read ☺
Shadow stacks are analyzed in the recent [BZP18]

Android 10 - Shadow stack

Protecting against code reuse in the Linux kernel with Shadow Call Stack
October 30, 2019

> *Google's Pixel 3 and 3a phones have kernel SCS enabled in the Android 10 update, and Pixel 4 ships with this protection out of the box. We have made patches available to all supported versions of the Android kernel and also maintain a patch set against upstream Linux.*

`https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html`

Intel Control-flow Enforcement Technology Specification

CET provides the following capabilities to defend against ROP/JOP style control-flow subversion attacks:

- *Shadow Stack – return address protection to defend against Return Oriented Programming,*
- *Indirect branch tracking – free branch protection to defend against Jump/Call Oriented Programming.*

`https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`

Outline

1. Memory corruption attacks
   - Format-string attacks
2. Mitigations
   - Stack canaries
   - Non-executable stack
   - Address-Space Layout Randomization
   - Control-flow Integrity
3. Learning resources

Learning resources (1/2)

For exploitation (there is a major overlap with already suggested resources), see:

- Pwn college: `https://pwn.college/` "*a first-stage education platform for students (and other interested parties) to learn about, and practice, core cybersecurity concepts in a hands-on fashion*"
- LiveOverflow's binary exploitation tutorials `https://www.youtube.com/watch?v=iyAyN3GFM7A&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN`
- Calle Svensson's streams, where he solves many challenges explaining step-by-step his reasoning/approach: `https://www.youtube.com/playlist?list=PLzzz0pYwY0M0u5daM96-QvHagA5v7FhLP`
- Adam Doupé's Pwnable.kr playlist `https://www.youtube.com/playlist?list=PLK06XT3hFPziMAZj8QuoqC8iVaEbrlZWh`

Learning resources (2/2)

- a very nice guide: "A First Introduction to System Exploitation" `https://research.checkpoint.com/2020/i-want-to-learn-about-exploitation-where-do-i-start/`
- wargame sites:
  - w3challs `https://w3challs.com/`
  - Pwnable.kr `https://pwnable.kr/`
  - ROPemporium `https://ropemporium.com/`
  - ...

See [SPWS13] for a survey of memory corruption attacks

References I

[ABEL05] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti.
Control-flow integrity.
In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[BB14] Erik Bosman and Herbert Bos.
Framing signals-a return to portable shellcode.
In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 243–258. IEEE, 2014.

[BBM+14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh.
Hacking blind.
In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

References II

[BZP18] Nathan Burow, Xinping Zhang, and Mathias Payer.
Shining light on shadow stacks.
arXiv preprint arXiv:1811.03165, 2018.

[CBP+15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross.
Control-Flow Bending: On the effectiveness of control-flow integrity.
In *USENIX Security Symposium*, pages 161–176, 2015.

[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.

References III

[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.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song.
Sok: Eternal war in memory.
In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.