
# ELF Section Docking
# Revisiting Stageless Payload Delivery

Dimitry Snezhkov,

X-Force, IBM Corporation

# Who -m

Dimitry Snezhkov, X-Force, IBM Corporation

- Research (Offense / Defense)
- Tooling Support

# Goals

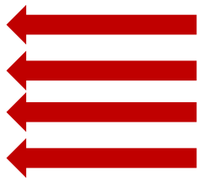
- Overview of static payload bundling mechanisms in Linux.
- Evolution of static payload embedding.
- Improving viability of static payloads in delivery.
- Binary compatibility of ELF sections as a unit of payload.
- ELF section docking: Payload attachment factory at adversarial sites.
- Detection, evasion successes and pitfalls of the docking approach
- ELFPack PoC demo

# Payload Delivery

| Dynamic | Static |
| --- | --- |
| • Generated at runtime<br>• Fetched at runtime from external (to loader) source<br><br>+ Light(er) loader<br>+ Less chance of detection due to absence of embedded payload<br>+ More flexibility<br><br>- More chance of detection with use<br>- More exposed loading mechanism<br>- More moving parts<br>- More detonation dependencies (environment).<br>- Long haul activation / dormancy issues. | • Bundled with delivery mechanisms<br>• Time-released<br><br>+ Less chances of detection due to close coupled variance<br>+ Less detonation dependencies (environment)<br>+ Less moving parts<br>+ Better activation / dormancy once deployed<br><br>- Heavier close coupled loader, greater size<br>- More chance of detection due to embedded payload<br>- Less flexibility (runtime awareness and variance)<br><br>it's a dial …  |

# Payload Delivery

| Dynamic | Static |
|---|---|
| • Empirically, more prevalent (short and long haul):<br><br>1. Deploy stager<br>2. Fetch payload<br>3. Load payload (maybe, on itself)<br>4. Maybe delete stager | • Empirically, less widespread in long haul implants<br>• Time-released<br><br>1. Deploy the bundle (maybe, on itself) |



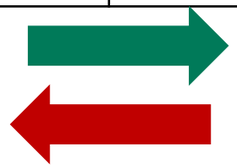
Dynamic is well understood

Can we improve **Static** delivery

# Payload Delivery Tradeoffs

- Why is static out of favor
- Can its traits be improved
- Can we turn downsides into an upside

| Desired Dynamic Traits | Undesired Static traits |
|---|---|
| + Less chance of detection due to absence of embedded payload<br>+ More flexibility | - Heavier close coupled loader, greater size<br>- More chance of detection due to embedded payload<br>- Less flexibility (runtime awareness and variance) |



# How We Embed

**Hex-binary inclusion compilation and linking:** Directly in default .data section via compiler

Manually or with tools like *bin2c* or

*xxd -i payload.bin > payload.h*

```
const data[3432] = {
    0x43, 0x28, 0x41, 0x11, 0xa3, 0xff,
    ...
    0x00, 0xff, 0x23
};
```

Easily traced at runtime debugging or static binary inspection

# How We Embed

**Hex-binary inclusion compilation and linking:** In a separate ELF section.

- Place payload data or certain variables in additional sections.
- Achieved with a compiler dependent mechanism. In gcc, it's done via __attribute__'s.

```
char stack[10000] __attribute__ ((section ("binstack"))) = {
    0x43, 0x28, 0x41, 0x11, 0xa3, 0xff,
    ...
    0x00, 0xff, 0x23 };
int init_data __attribute__ ((section ("bindata"))) = 0;

main()
{
    /* Initialize stack pointer */
    init_sp (stack + sizeof (stack));

    /* Initialize initialized data */
    memcpy (&init_data, &data, &edata - &data);
}
```

Can be traced at runtime debugging or static binary inspection.

# How We Embed

**Linker-binary inclusion:** Assembler and linker specific directives.

Assembler dependent .incbin-like directive can create a section and embed a payload.

Tools:

- gcc -c payload.s

or

- ld -r -b payload.bin -o payload.o

Note: fully functional payload file. Path to create “fat” binaries for packing.

Retrieval in code can be done as follows:

```
.section .bindata

.global payload_start
.type payload_start, @object

.section .binddata
.balign 64

payload_start:
    .incbin "payload.bin"
    .balign 1
payload_end:
    .byte 0
```

```
int main(void) {
    extern uint8_t payload_start;
    uint8_t *ptrPayload = &payload_start;
    ...
}
```

# How We Embed

**Linker-binary inclusion:** Assembler and linker specific directives (Cont.)

More ergonomic tools exist

- *INCBIN* from @graphitemaster, same idea.

In-code solution to construct multi-sectional ELF payload may be as follows:

Note PROGBITS directive, will be important.

```
/* Raw image data for all embedded images */
#undef EMBED
#define EMBED( _index, _path, _name )
        extern char embedded_image_ ## _index ## _data[];
        extern char embedded_image_ ## _index ## _len[];
        __asm__ ( ".section \".rodata\", \"a\", " PROGBITS "\n\t"
                  "\nembedded_image_" #_index "_data:\n\t"
                  ".incbin \"" _path "\"\n\t"
                  "\nembedded_image_" #_index "_end:\n\t"
                  ".equ embedded_image_" #_index "_len, "
                        "( embedded_image_" #_index "_end - "
                        "  embedded_image_" #_index "_data )\n\t"
                  ".previous\n\t" );
EMBED_ALL

/* Image structures for all embedded images */
#undef EMBED
#define EMBED( _index, _path, _name ) {
        .refcnt = REF_INIT ( ref_no_free ),
        .name = _name,
        .data = ( userptr_t ) ( embedded_image_ ## _index ## _data ),
        .len = ( size_t ) embedded_image_ ## _index ## _len,
},
static struct image embedded_images[] = {
        EMBED_ALL
};
```

# How We (Better) Embed

**Compiler / linker-based payload are not ideal.**

The process of embedding in code is tightly coupled to the creation of payload loader.

- Challenges with payload format changes
- By default, data carrying section have PROGBITS flags set on it, and it will be PT_LOAD'ed into memory by the OS loader by default.

**We do not want this (Linking ➔ Detection)**

There are tradeoffs 

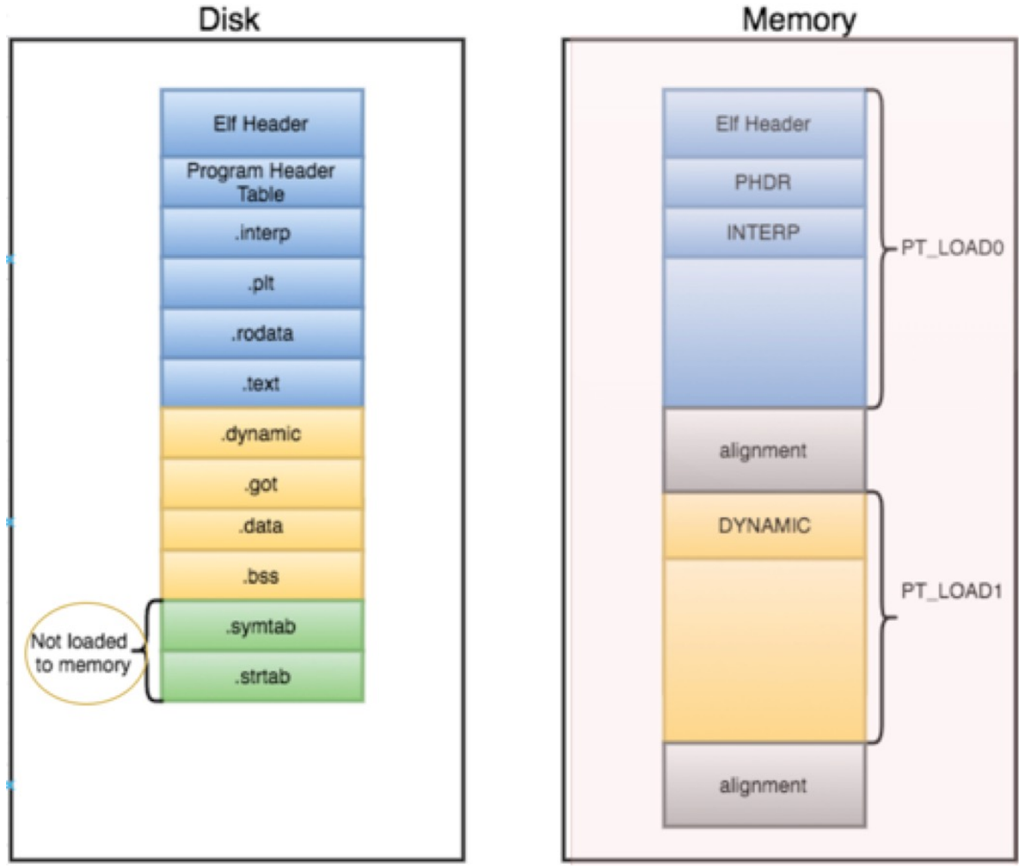
# How We (Better) Embed

**ELF sections and load flags**

*Type of section and flags set on the new section determine whether OS loader loads it in the memory upon executable launch.*

Some sections are loaded automatically by default, others are not.

**Offense can take advantage of that!**



# How We (Better) Embed: Take 2

**Avoiding default OS loader actions**

We can:

- Avoid setting flags on sections that assume default loading in memory.
- Use a different type of section that does not load in memory.
- As an example – SHT_NOTE type, from ELF docs:

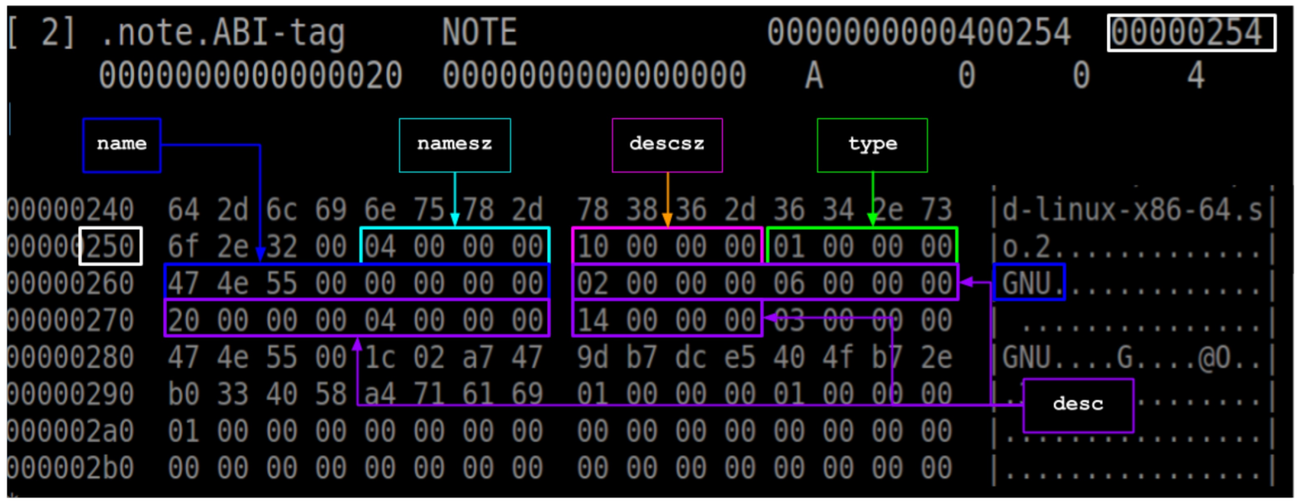
> A vendor or system engineer might need to mark an object file with special information that other programs can check for conformance or compatibility. Sections of type `SHT_NOTE` and program header elements of type `PT_NOTE` can be used for this purpose.

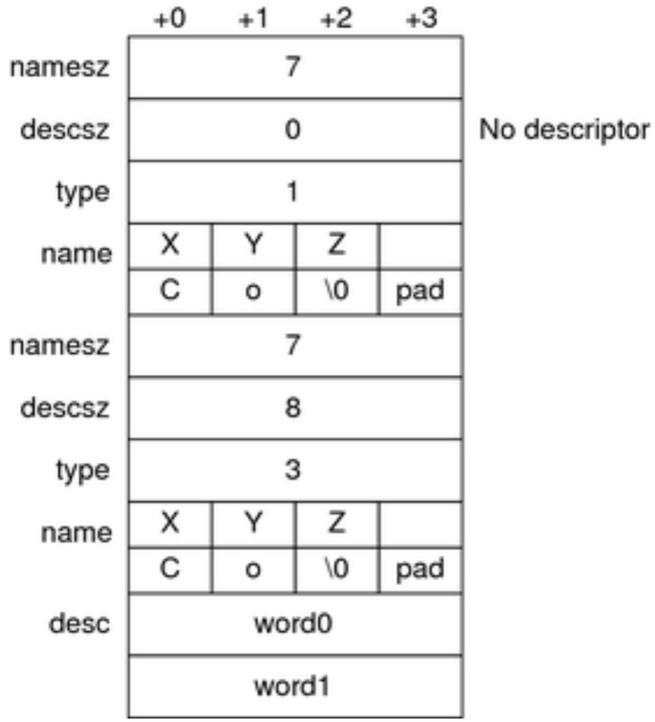
# How We (Better) Embed: Take 2

**Avoiding default OS loader actions**

SHT_NOTE is widely used in Linux system binaries:

```
$ readelf --sections /bin/tar | grep NOTE
  [ 2] .note.gnu.bu[...] NOTE             00000000000002c4  000002c4
  [ 3] .note.ABI-tag     NOTE             00000000000002e8  000002e8
```



# ELF Section Docking

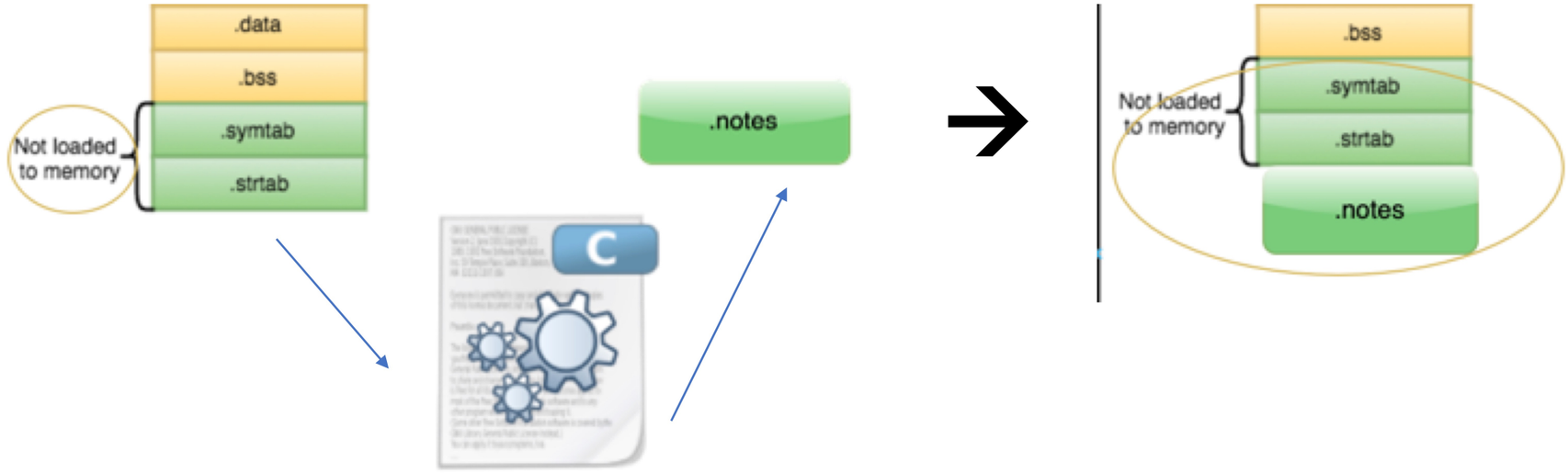
**Compiler is a problem: Decoupling payloads**

So far, we have:

- Created a dormant section in ELF image (in code)
- Avoided loading it in memory by the OS loader.

However:

- Section ←→ structure of the final ELF
- Tight relationships of memory addresses from **the loader code**



# ELF Section Docking

**Compiler is a problem: Decoupling payloads**

What if we:

- Create an ELF section with embedded payload **outside** of the loader compilation workflow
- Attach that section to a loader binary later

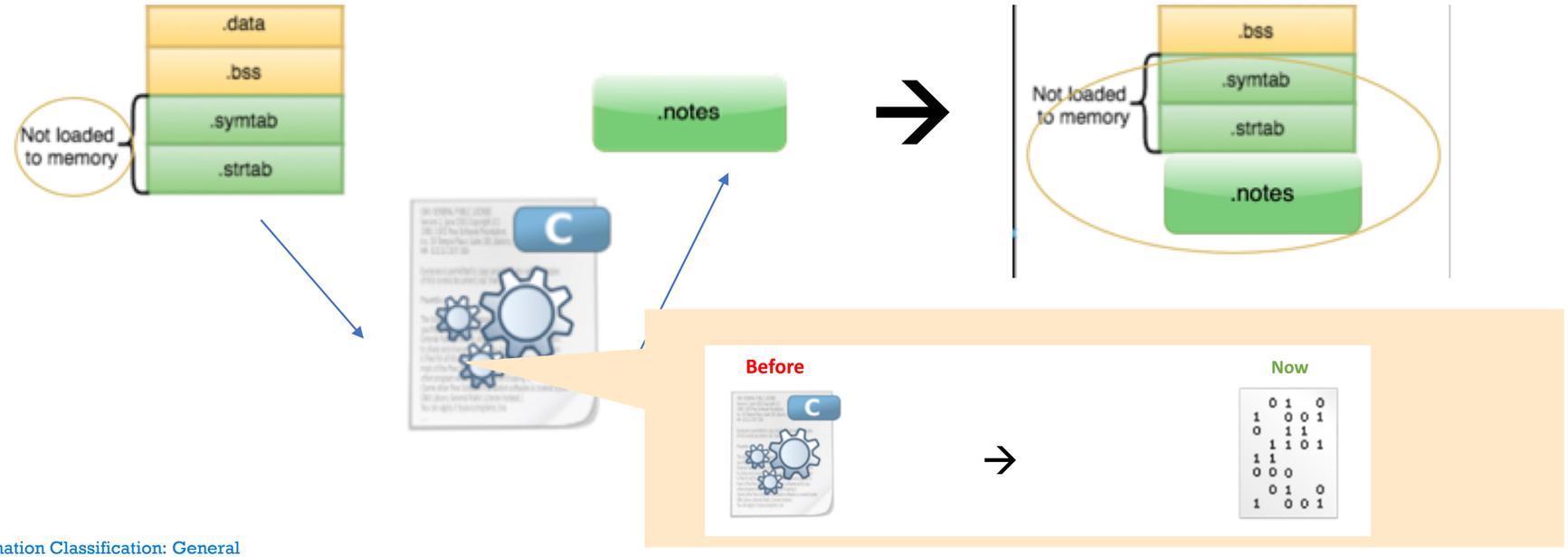
This would:

- Break the address offset relationship of the loader code with the section.
- Teach the loader how to find and load its ***foreign*** data section, effectively "*docking*" a standalone payload to a loader in a loosely coupled manner.

# ELF Section Docking

**Compiler is a problem: Decoupling payloads, Avoiding OS loader**

- Loader should not be entangled with payload semantics
- Loading and executing binary payload without modifying loader code to tailor to new payloads via binary section compatibility.
- Loading without using OS loader ld.so (ELF loader) which is loading payload in memory automatically.

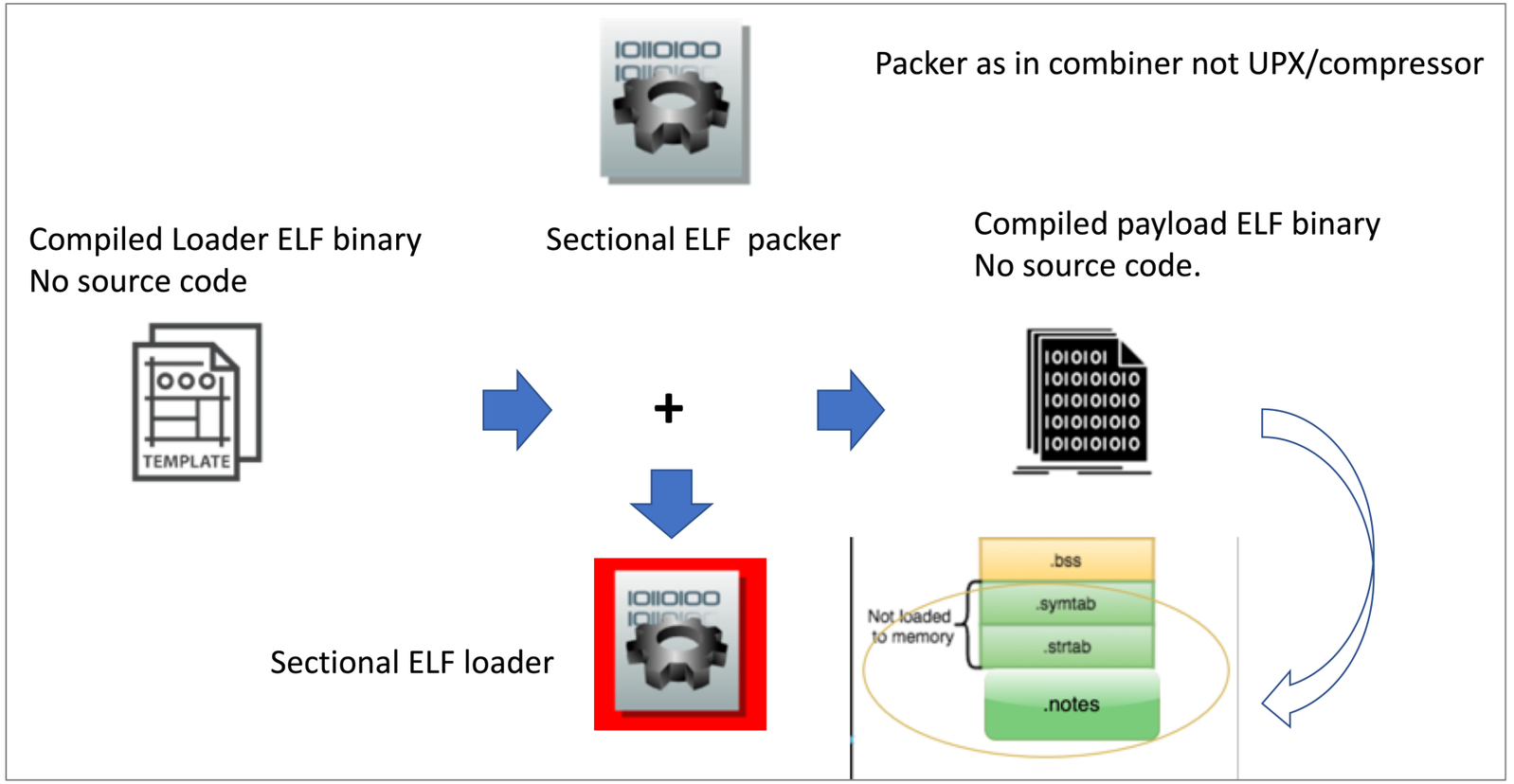



**Achieves ABI compatible In-field payload (re-)attachment.**

# ELF Section Docking

**Binary Compatibility at Section level**

- Injector/bundler which will introduce a payload section to the loader without either one operating at code level, only binary compatibility
- Loader is aware *how* to load a payload section but not what the payload is.

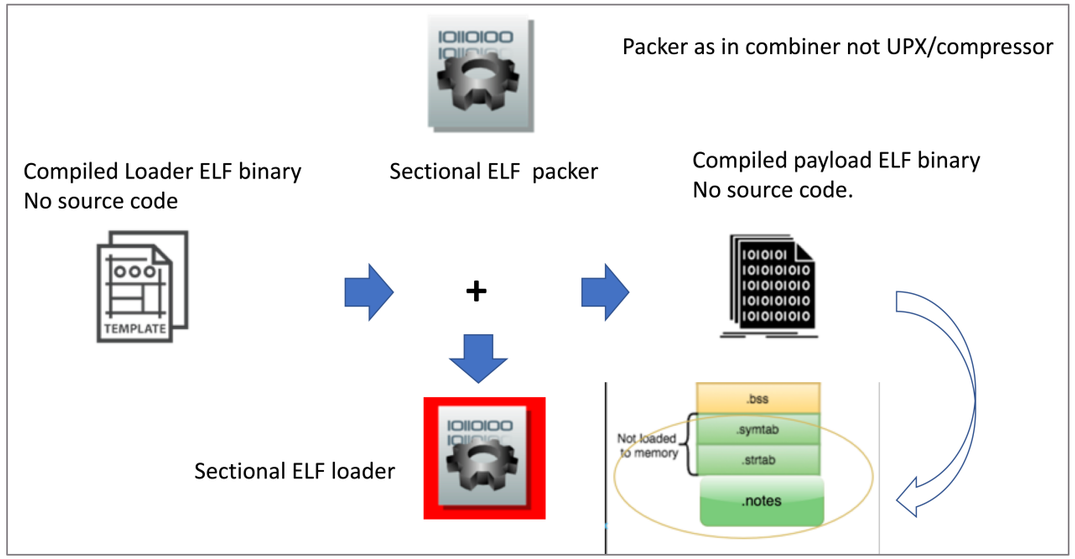


# ELF Section Docking

**Static Elf loader:**

- Shipped on its own
- Devoid of payloads
- Only mechanisms to load a section on demand and bootstrap the payload from it.

**Possible Wins**

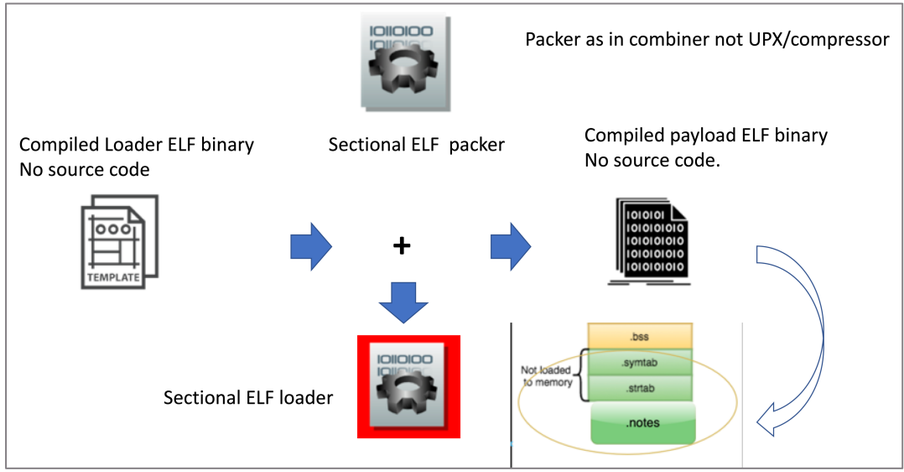



**Sectional Payload:**

- Created separately
- Bundled with loader *at any time* as a static stage
- Better dormancy control with an injector.
- Better packing. No overhead on detection for conventional packer processing and code. In memory – not tmpfs for unpacks. Fat binaries possible (multiple sections).
- Be a full ELF executable itself if needed

**Injector** can broker attachment of sections from several binaries (dormant stages) to construct a section and inject into the loader.

# ELFPack




## Sectional ELF Injector/Packer:

- Streamlined payload generation pipeline
- In field payload to loader attachment without compiler

## Sectional ELF Loader:

- Loads full ELFs or shellcode from reading and parsing its own binary.
  Tracing does not see *mprotect()*'s on mapping into memory and loading
- Airgapped separation between where the payload is and how it's loaded.
- Ability to accept and forward arguments to sectional payloads

## Binary payload in section

- Can be a fully functional ELF binary with much less constraints (3rd party tooling, linking intact).
- Can be uniquely obfuscated without regard to space (.NOTE records are variable size for example)
- Can be memory-resident or extracted to FS or run as part of a table of contents (fat payload loader).
- Does not need to be relocated when preparing for execution.
- Cross-attachment binary evasion chain: **Loader A can read Loader B's payload.**

# ELFPack – Loader

**Option A** : SYS_Memfd_create ()

- Done with libreflect but may be done with Zombieant pre-loader
- More detectable at levels:
  - anonymous file in /proc/self/fd/
  - *uses sys_memfd_create* ( syscall #319 )
- Does fork/exec, BPF tracing for execve() will record.

**Option B:** User land Exec (https://grugq.github.io/docs/ul_exec.txt)

- Done with libreflect for now. Nice interface.
- Hollows out the loader and overlays with payload.
- No sys_enter_exec /sys_exit_exec calls. BPF tracing for execve() not catching
- Downside: you cannot daemonize via loader (loader memory is gone on exec image overlay)
  but the payload can daemonize itself when launches:
  the beauty of shipping ELF binaries vs. shipping shellcode ☺

# ELFPack: Detect: Binwalk

Raw Payload (mettle)

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 64-bit LSB shared object, AMD x86-64, version 1 (SYSV)
184267        0x2CFCB         bix header, header size: 64 bytes, header CRC: 0x83E0FE, created: 2043-02-09 12:09:36, image size
: 18565 bytes, Data Address: 0xED48896B, Entry Point: 0x58741645, data CRC: 0x85E47411, compression type: none, image name: ""
375762        0x5BBD2         bix header, header size: 64 bytes, header CRC: 0x488D94, created: 1989-08-21 13:13:36, image size
: 4754292 bytes, Data Address: 0x2408B930, Entry Point: 0x48, data CRC: 0x89EF4889, image name: "("
655552        0xA00C0         Base64 standard index table
660102        0xA1286         Unix path: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/gam
es:/system/bin:/system/sbin:/system/xbin
668544        0xA3380         Base64 standard index table
680416        0xA61E0         Base64 standard index table
682452        0xA69D4         Unix path: /usr/bin/ntlm_auth
688177        0xA8031         PEM certificate
691424        0xA8CE0         DES SP2, little endian
691680        0xA8DE0         DES SP1, little endian
694019        0xA9703         PEM RSA private key
694240        0xA97E0         SHA256 hash constants, little endian
694656        0xA9980         Base64 standard index table
704037        0xABE25         Unix path: /sys/devices/system/cpu/cpu%d/cpufreq/cpuinfo_max_freq
704994        0xAC1E2         Unix path: /sys/class/net/%s/speed
705422        0xAC38E         Unix path: /dev/disk/by-uuid
706607        0xAC82F         Copyright string: "Copyright 1995-2013 Jean-loup Gailly and Mark Adler "
709775        0xAD48F         Copyright string: "Copyright 1995-2013 Mark Adler "
712640        0xADFC0         CRC32 polynomial table, little endian
716736        0xAEFC0         CRC32 polynomial table, big endian
724288        0xB0D40         Unix path: /usr/local/bin:/bin:/usr/bin
727202        0xB18A2         Unix path: /var/run/nscd/socket
```

Sectioned Payload (mettle)

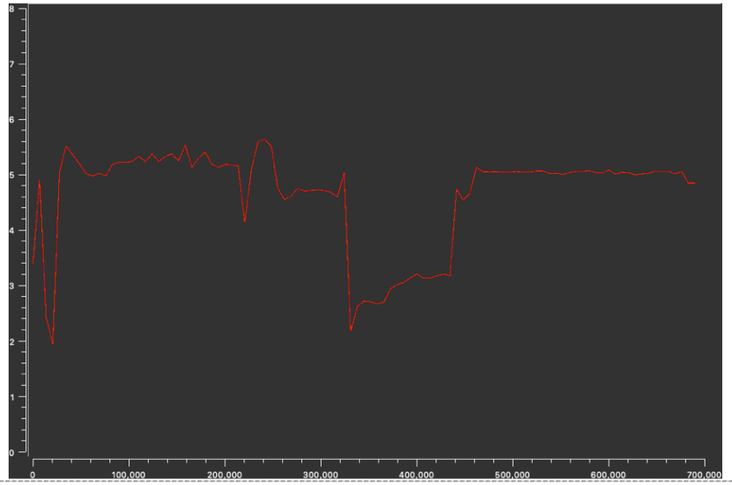
```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 64-bit LSB shared object, AMD x86-64, version 1 (GNU/Linux)
92200         0x16828         Unix path: /home/dev/Code/elfpack/src/elfldr.cpp
95464         0x174E8         Unix path: /home/dev/Code/elfpack/src/elfldrlib.cpp
```

# ELFPack: Detect: Biwalk: Entropy

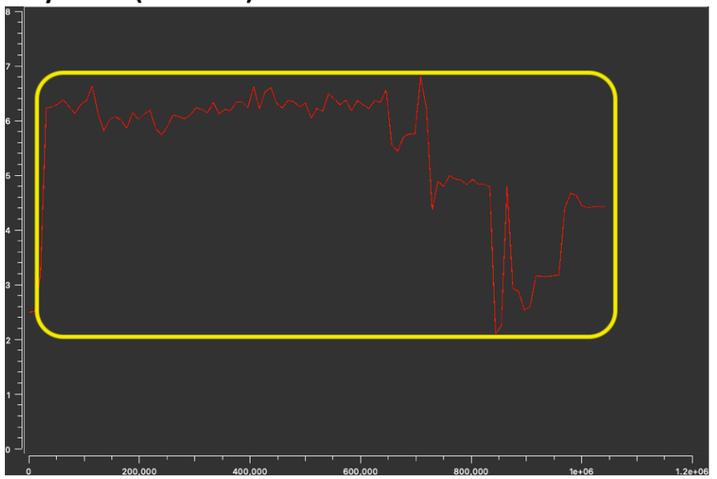
Raw Payload (mettle)

Sectioned Payload (mettle)

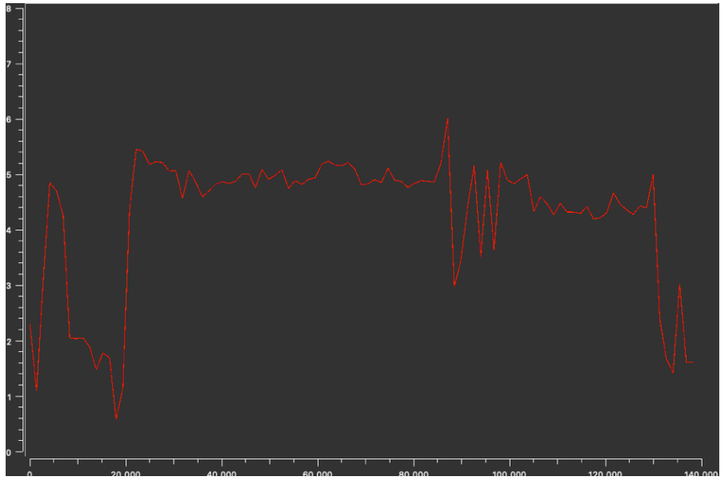
ELFPacker



Payload (Mettle)



ELFLoader



ELFPack'd Loader + Payload

# ELFPack: Detect: BPF + YARA

More detection and evasion

## BPF filter based

Tracepoints -> syscalls:

**Sys_enter_memfd_open**
**Sys_exit_memfd_open**
**Sys_enter_exec***

## YARA static scan ELF

```
(venv) dev@devpc6:~/Code/elfpack/aux/triage$ python3 elfpack_yar.py
=== SHT_NOTES Sections ===
.note.gnu.build-id        :     36 bytes
.note.ABI-tag             :     32 bytes
.note.gnu.buf[...]        : 1042180 bytes
.note.gnu.buf             :     40 bytes
.00000000: 04 00 00 00 10 00 00 00  01 00 00 00 47 4E 55 00  ............GNU.
00000010: 00 00 00 00 03 00 00 00  02 00 00 00 00 00 00 00  ................
None
.
----------------------------------------------------------------------
Ran 2 tests in 0.027s

OK
```

# ELFPack Demo

# Summary

- Section docking presents desired features for payload delivery
- Static vs. dynamic payload loading is a dial not an either or.
- Overcome limitations of packers for in-memory unwrap and detection
- Detect ELF packing at runtime and static.
- Overcome detections with packing and encryption.

# Q&A?

Code: https://github.com/xforcered/elfpack

Thanks!