# Phase I: Sectional ELF loader with binary payload attachment

Possible Interesting Uses:

- Air gapped execution
- Generalized loader, split from payload semantics (BYOP)
- Drive by payload attachments to loader as in payload generation pipeline.
- Fat payload binaries
- Avoid use of packers (that create extra signatures and generate artifacts)
- Payload obfuscation
- Payload obfuscation key protection options
- BPF execution tracing resistance.

# Phase II: In-memory Payload Facilites

- SYS_Memfd_create ()
- User land Exec
- Resistance to BPF tracing sensors

# How we embed payloads

Include binary file in C code

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

const int data_length = 3432;
```

- Bin2c
- xxd -i mybinary > myheader.h and include header

- Keeping payloads in .text is not ideal
- Can keep them in a separate section if needed:
  - with __attribute__s
- Loading is transparent and can be traced to memory address (offset from .text)

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 }
int init_data __attribute__ ((section ("INITDATA"))) = 0;

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

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

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

# How we embed payloads

Include binary file into a section with linker  (similar to user defined resource in Windows)

Compile into a section

```
.section .bindata

.global imrdls_start
.type imrdls_start, @object

.global imr_SW_DL_start
.type imr_SW_DL_start, @object

.section .bindata
.balign 64
imrdls_start:
imr_SW_DL_start:
    .incbin "file.bin"
    .balign 1
imr_SW_DL_end:
    .byte 0
```

Reference from C

```
int main(void) {
    extern uint8_t imrdls_start;
    uint8_t *ptrToExpectedDL = &imrdls_start;

    for(int i = 0; i < 135; i++)
    {
        printf("0x%02x ", ptrToExpectedDL[i]);
        if((((i + 1) % 15) == 0)) printf("\n");
    }

    return EXIT_SUCCESS;
}
```

Options A. Via Compiler: gcc -c thing.s

Option B. Via Linker: ld -r -b binary -o binary.o foo.bar # then link in binary.o

# How we embed payloads

Include binary file with C code and linker but more ergonomically – INCBIN from @graphitemaster

Compile into a section

```
#define INCBIN_PREFIX b
#define INCBIN_OUTPUT_SECTION ".rodata"
#include "incbin.h"

#ifdef __cplusplus
extern "C" {
#endif

#define STRINGIZE(x) #x
#define STRINGIZE_VALUE_OF(x) STRINGIZE(x)

INCBIN(BD, STRINGIZE_VALUE_OF(BDATA));
// INCTXT(TD, STRINGIZE_VALUE_OF(TDATA));

#ifdef __cplusplus
}
#endif

#endif //BUNDLER_BUNDLES_H
```

Reference from C

```
#include "bundler.h"

int main(int argc, char **argv){
    printf("Binary: size: %d data start loc: %p \n",
           bBDSize,
           bBDData
           );
```

# How we embed payloads

Can also do via inline assembly but more complex inclusions are less flexible

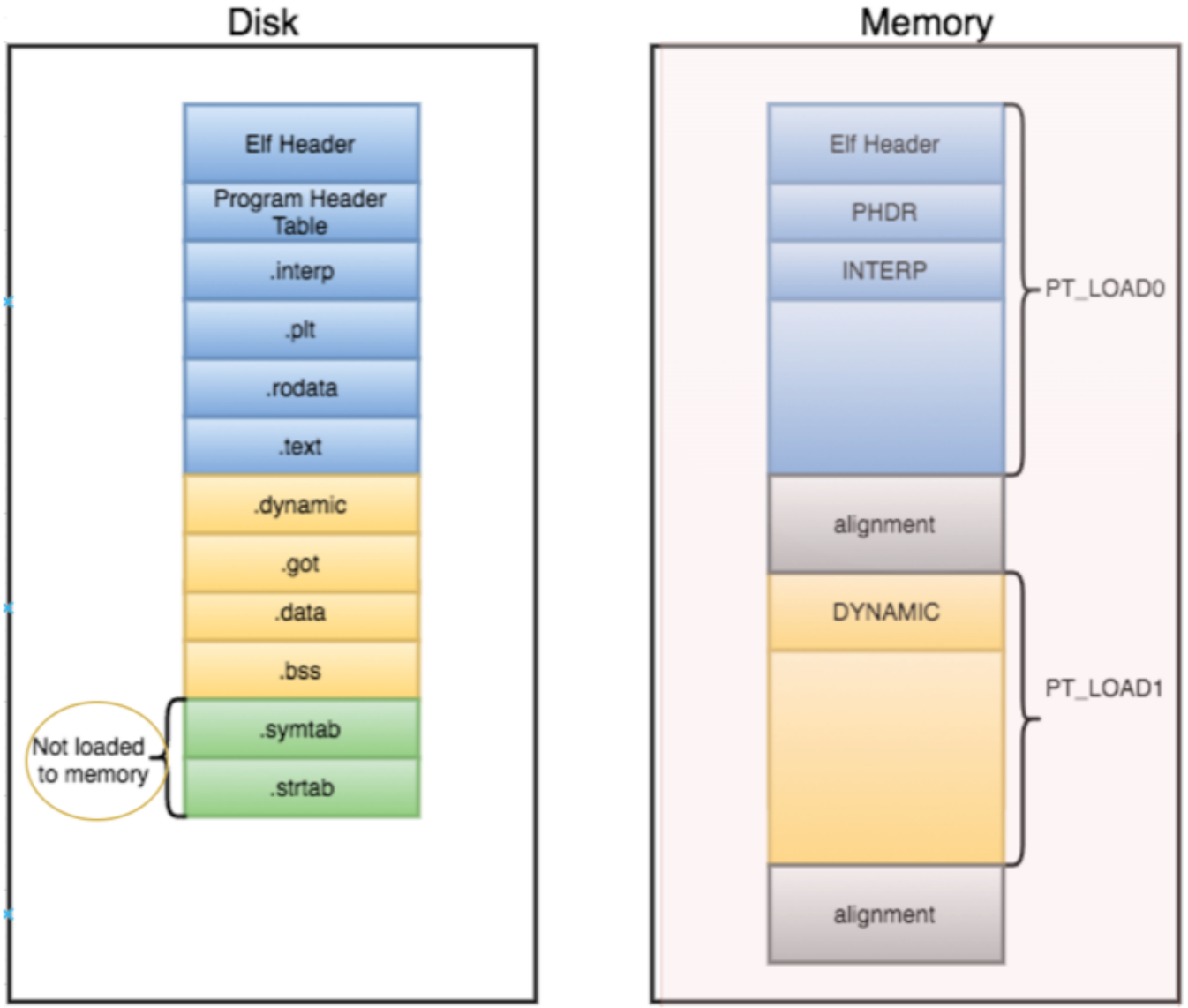
Tightly coupled
What about payload format changes
**What about PROGBITS – PT_LOAD'ed**
**You like dis?** →

.

```
/* 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
};
```

# What about **PROGBITS** ? … – PT_LOAD'ed

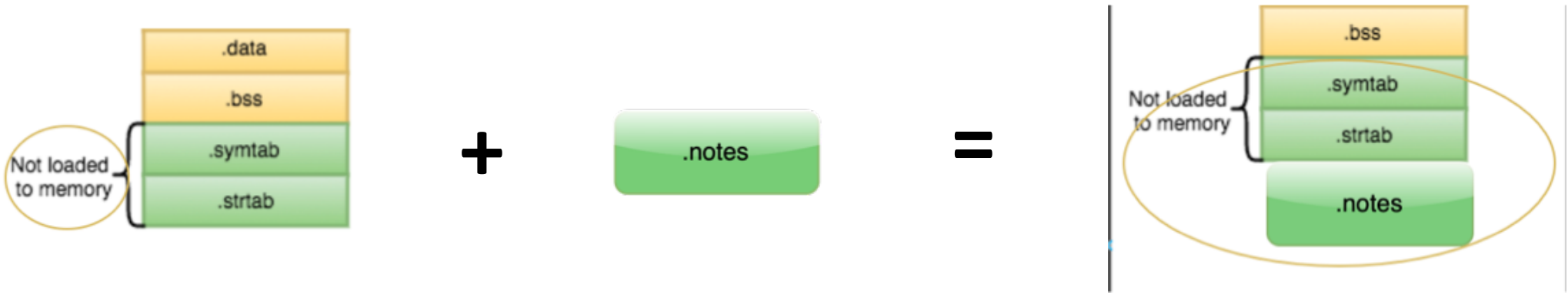


# How we embed payloads: Take 2

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.

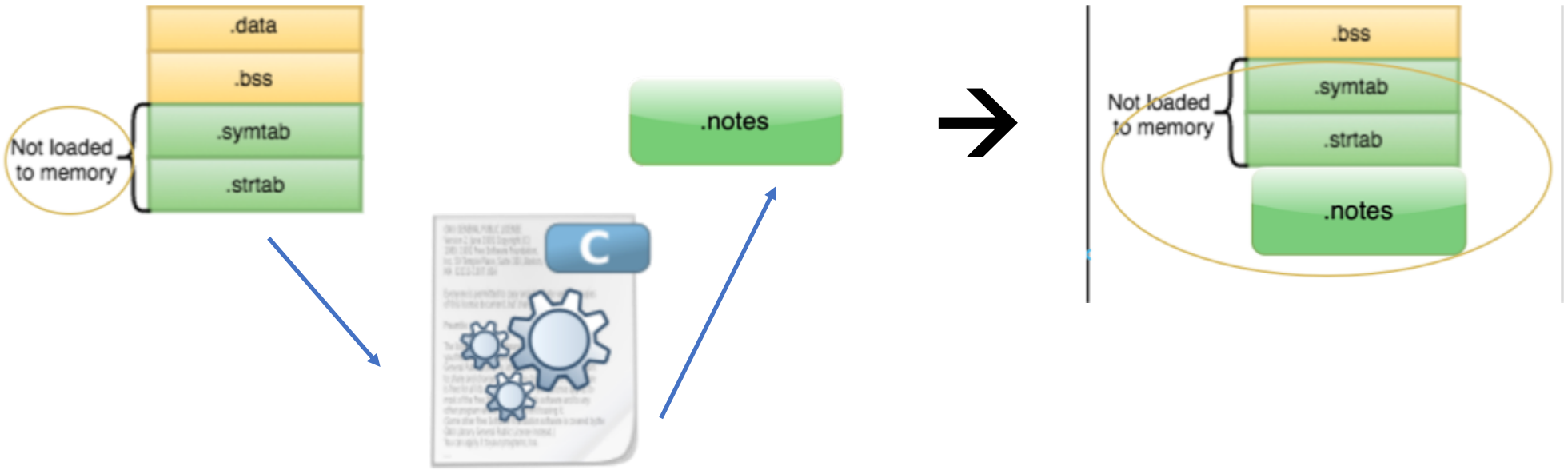
```
dev@pc0667:~$ readelf --sections /bin/tar | grep NOTE
  [ 2] .note.gnu.bu[...] NOTE             00000000000002c4  000002c4
  [ 3] .note.ABI-tag     NOTE             00000000000002e8  000002e8
dev@pc0667:~$ readelf -p .note.ABI-tag  /bin/tar

String dump of section '.note.ABI-tag':
  [     c]  GNU
```



```
/* 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
```

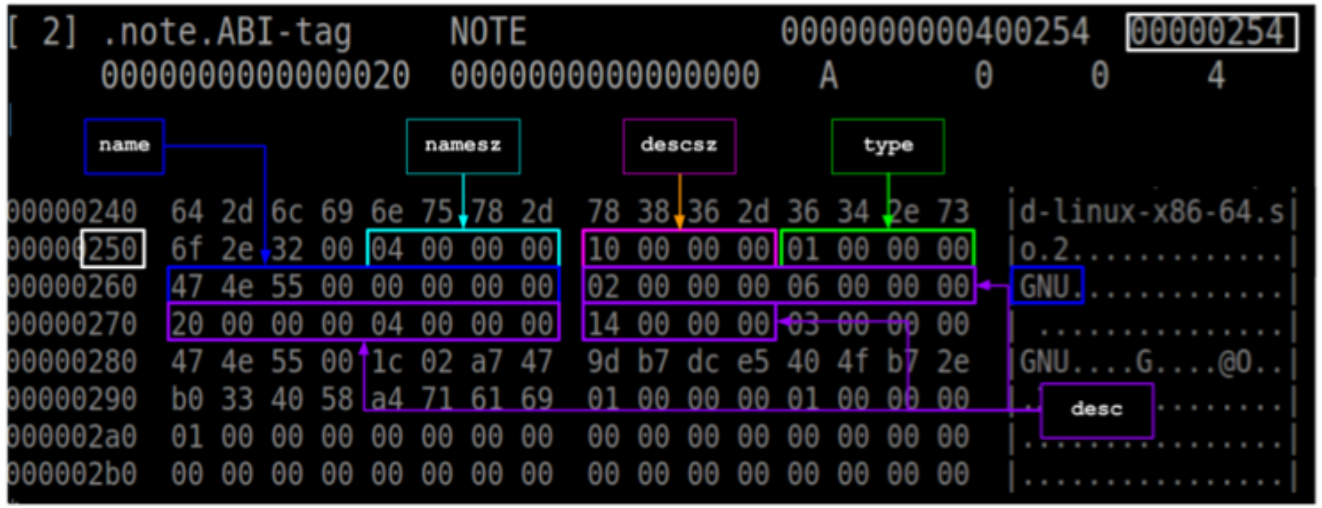
Fine, I will change this to .NOTES. Happy?

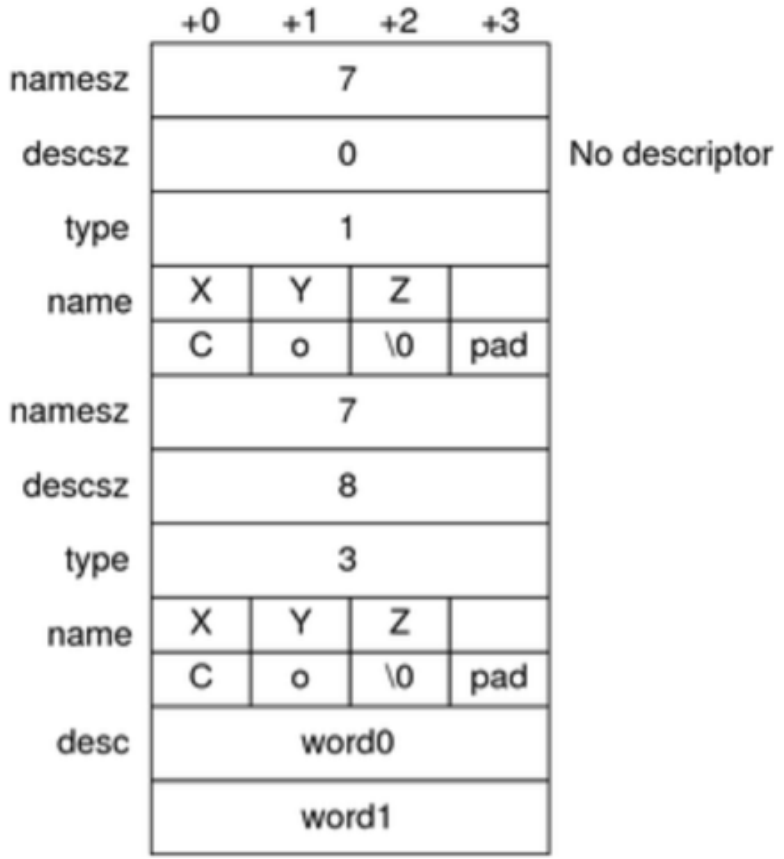


# Structured Sections

.notes

=



# How we load payloads

- Loader should not be entangled with payload semantics
- Can we load and execute payload :
  - Without modifying loader code, like at all?
  - Without ld.so (ELF loader) loading segments of payload in memory automatically.
  - In-field payload (re-)attachment. (maybe?)

## Loader / payload relationship

**Before**

→

**Now**

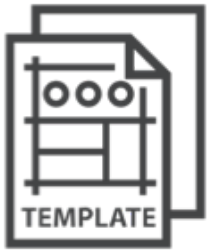
# Drive-by ELF Binary Section Attachment

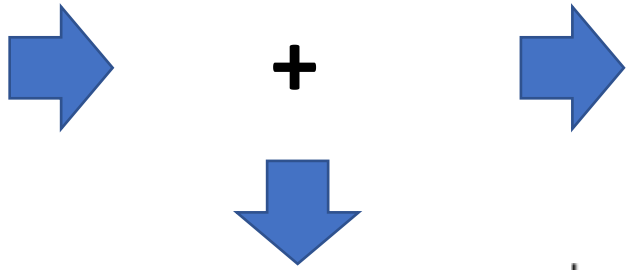
Packer as in combiner not UPX/compressor

Compiled Loader ELF binary
No source code

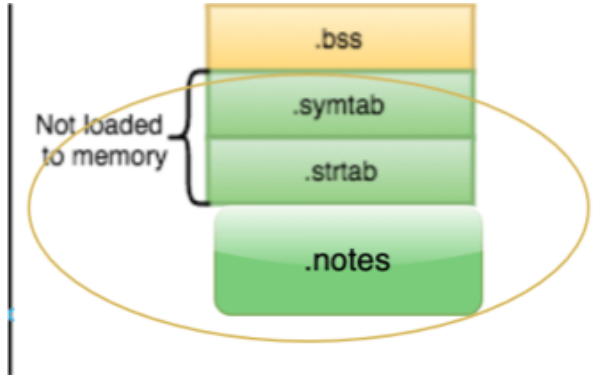
Sectional ELF packer

Compiled payload ELF binary
No source code.



Sectional ELF loader



# Drive-by ELF Binary Section Attachment

Sectional ELF packer

Wins:
- Agnostic loader to payload proxy
- Streamlined payload generation pipeline
- In field payload to loader attachment without compilers if needed

Sectional ELF loader

Wins:
- Agnostic to payload
- Loads full ELFs not shellcode (more possibilities) from reading and parsing its own binary
- If you need shellcode you can create a running elf out of it (e.g. mettle)
- Tracing does not see mprotect()'s
- Airgapped separation between where the payload is and  normal .DATA arrays.
- This achieves abstraction for tracers.
- Arguments to payloads

Binary Payload

Wins:
- Payload is a fully functional program with less constraints, data, segments LDD intact.
- It can be uniquely obfuscated without regard to space (.NOTE records are variable size)
- It can be extracted to FS or run as part of a table of contents (fat payload loaders).
- It does not need to be relocated, can be chained to other loaders.
  - Example of evasion: Loader A reads Loader B's payload ☺

# Drive-by ELF Binary Section Attachment

Sectional ELF packer

- For now – XOR'd payload but AES may be implemented.
- XOR key metadata stored out in out of band watermark.
- XOR keys are not disclosed.
- Additional XOR'd data obfuscation with LR bit shifts by position

Sectional ELF loader

- For now – XOR'd payload, but AES may be implemented.
- XOR key metadata is mined in out of band watermark.
- Separation of time of loader launch != time of payload load if needed.
- Facility for daemonization
- Binwalk does not see payload, can't carve

# Phase II: In-memory Payload Facilities

**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 I think)
- 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 kaput on overlay)
  but the payload can daemonize itself when launches:
  the beauty of shipping ELF binaries vs. shipping shellcode ☺

# Demo Time