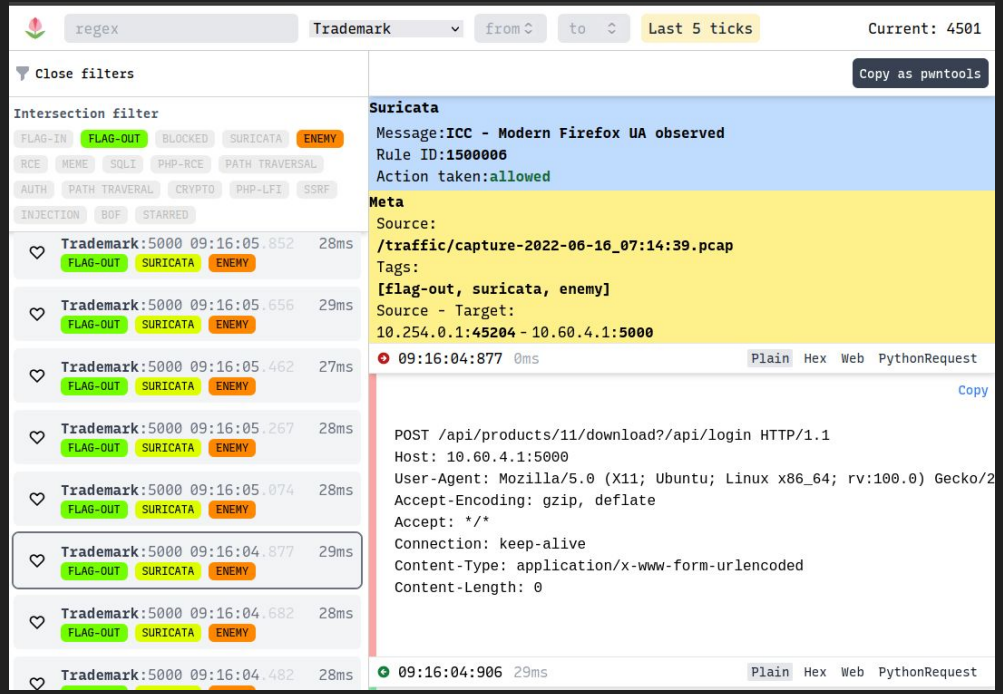
eBPF: Utilizzi nelle CTF Attack/Defense

Alessandro Zanier
Gabriel Prostitis

Analisi del traffico



Fingerprinting Application-level

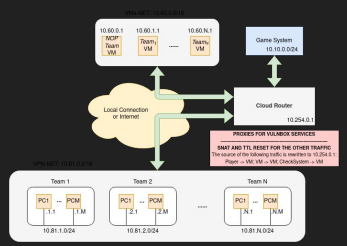
- Basato sui dati trasmessi
- Header HTTP: **User-Agent**, **Accept-Encoding**, ordine
- Alfabeto e lunghezza delle credenziali generate
- Numero di flag esfiltrate
- Non utilizzo di **recv*** e **send*after**

```
if request.headers["User-Agent"] ≠ "CHECKER":
  return DROP

if request.form["username"] == "prova123"
  return DROP

...


if client_payload.count(b"\n") > 1:
  return DROP

if len(FLAG_REGEX.findall(server_payload)) > 1:
  return DROP
```

Fingerprinting “Low-level”

TCP header format[17]

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source Port																Destination Port														
4	32	Sequence Number																														
8	64	Acknowledgement Number (meaningful when ACK bit set)																														
12	96	Data Offset				Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window														
16	128	Checksum																Urgent Pointer (meaningful when URG bit set)[18]														
20	160	(Options) If present, Data Offset will be greater than 5. Padded with zeroes to a multiple of 32 bits, since Data Offset counts words of 4 octets.																														
⋮	⋮																															
56	448																															

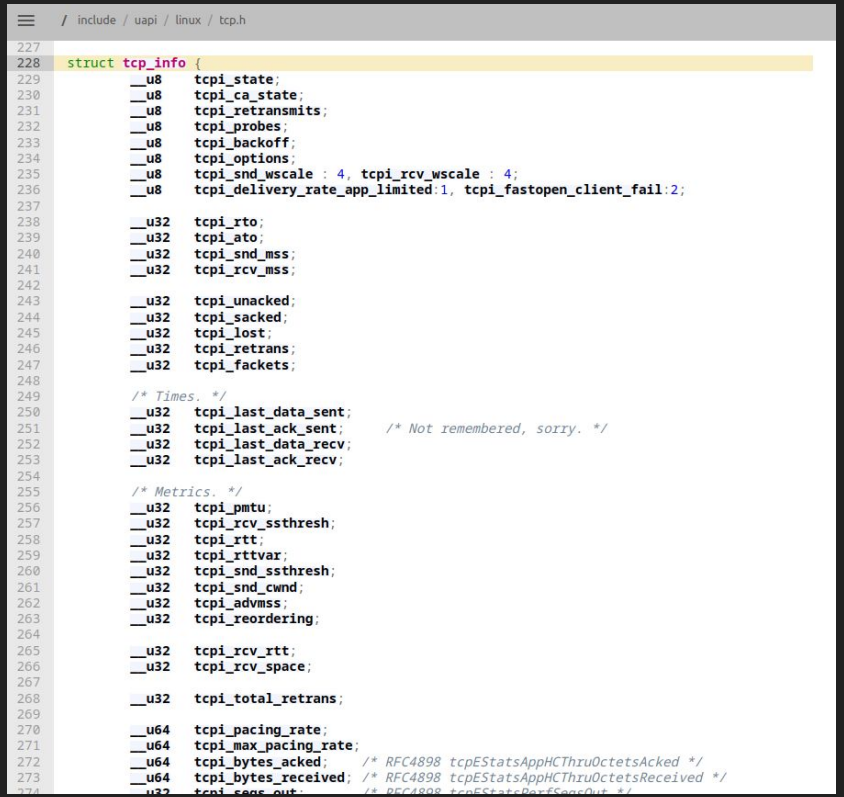


Unione

- Fingerprinting application-level facile da implementare
- Fingerprinting low-level più complesso
- Posso sfruttare le due tecniche contemporaneamente?

Unione

- AF_PACKET/nfqueue
- TCP_INFO sockopt
- eBPF



Cos’è eBPF?

- **eBPF** (Extended Berkeley Packet Filter), successore di cBPF, è una tecnologia che permette di eseguire, entro certi limiti, dei programmi lato kernel.

  Viene principalmente utilizzato per estendere le normali funzionalità del kernel stesso.

  Per interagire con il sottosistema si utilizza la syscall **bpf** (#321).

- **https://docs.ebpf.io/**
- **https://man7.org/linux/man-pages/man2/bpf.2.html**

Cos’è eBPF?

- I programmi **eBPF** possono essere utilizzati per vari scopi, per questo motivo esistono diverse tipologie di programmi.

  Inoltre il kernel Linux può limitare o modificare il comportamento dei programmi stessi in base al **tipo di programma**, ai **permessi** dell’utente che ha caricato il programma stesso o, in generale, alla **configurazione del sistema**.

- Per utilizzare eBPF il programma potrebbe avere bisogno delle seguenti capability:
  - **CAP_BPF**
  - **CAP_NET_ADMIN**
  - **CAP_PERFMON**

Program Loading

- I programmi non sono costituiti da istruzioni native del processore su cui vengono eseguiti, bensì da istruzioni specifiche della **eBPF Virtual Machine**.

- La VM dispone di **10 registri general-purpose** (R0-R9) e di un registro dedicato al **frame pointer** (R10).

- Supporta **otto classi di istruzioni**, che consentono un ampio spettro di operazioni.

```
enum bpf_prog_type type;
const struct bpf_insn *insns;
int insn_cnt, pfd;
const char *license;

union bpf_attr attr = {
      .prog_type = type,
      .insns     = ptr_to_u64(insns),
      .insn_cnt  = insn_cnt,
      .license   = ptr_to_u64(license),
      .log_buf   = ptr_to_u64(bpf_log_buf),
      .log_size  = LOG_BUF_SIZE,
      .log_level = 1
};

pfd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```

Program Types (examples)

- Network program types (**CAP_NET_ADMIN**)
  - **BPF_PROG_TYPE_SOCKET_FILTER**
  - **BPF_PROG_TYPE_XDP**

- Tracing program types (**CAP_PERFMON**)
  - **BPF_PROG_TYPE_KPROBE**
  - **BPF_PROG_TYPE_TRACEPOINT**
  - **BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE**

I programmi sono solitamente associati ad eventi per esempio la ricezione di un pacchetto o l'esecuzione di una determinata funzione lato kernel.

Ad ogni tipo di programma vengono passati dati diversi (aka: **context**)

Maps

- Programmi eBPF possono scambiare informazioni tra di essi o con processi in user space tramite mappe.

- Esistono varie tipologie di mappe:
  - **BPF_MAP_TYPE_HASH**
  - **BPF_MAP_TYPE_ARRAY**
  - **BPF_MAP_QUEUE**
  - **BPF_MAP_STACK**

```
enum bpf_map_type map_type
unsigned int key_size
unsigned int value_size
unsigned int max_entries
int mfd;

union bpf_attr attr = {
.map_type    = map_type,
.key_size    = key_size,
.value_size  = value_size,
.max_entries = max_entries
};

mfd = bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
```

Helper functions & KFuncs

- Funzioni definite dal kernel e utilizzabili dai programmi

  Alcune di queste sono utilizzabili solo da certi tipi di programmi
  Esempi:

  - **bpf_map_{lookup/update/delete}_elem**
  - **bpf_map_{push/pop}_elem**
  - **bpf_get_current_uid_gid**
  - **bpf_get_current_pid_tgid**
  - **bpf_skb_{store/load}_bytes**
  - **bpf_xdp_{store/load}_bytes**

Fingerprinting w/ eBPF: Idea

- Mappa BPF per passare i dati usando IP/porta come chiavi
- Programma sul del socket in ascolto (**BPF_PROG_TYPE_SOCK_FILTER**)
  - **setsockopt** se posseggo l'fd del socket
  - **pidfd_getfd** se voglio "rubarlo" senza cooperazione del processo target*

*https://blog.cloudflare.com/tubular-fixing-the-socket-api-with-ebpf/, https://lwn.net/Articles/808997/

Fingerprinting w/ eBPF: Idea

In questo modo:

- Vedo solo i pacchetti diretti alla porta del socket a cui sono attaccato
- Garanzia di vedere i pacchetti “prima” che arrivino al socket
- Dati facilmente raggiungibili da userland tramite le BPF maps
- Non vedo i pacchetti in uscita (ci servono?)

Fingerprinting w/ eBPF: Tempo

Non esiste un vero helper che ritorni il tempo come UNIX timestamp

- La cosa più vicina è **bpf_ktime_get_tai_ns**
- Differisce di un offset che di fatto è costante

Fingerprinting w/ eBPF: Esempio

```
struct tcp_fprint_key {
    __u8 ip[16];
    __u16 port;
};

struct example_fprint {
    __u64 time;
    __u8  ttl;
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 65536);
    __type(key, struct tcp_fprint_key);
    __type(value, struct example_fprint);
} example_fingerprint SEC(".maps");
```

Fingerprinting w/ eBPF: Esempio

```c
SEC("socket")
int socket_handler(struct __sk_buff *skb)
{
    struct tcphdr tcp;
    struct example_fprint fprint;
    struct tcp_fprint_key key;

    __u64 time = bpf_ktime_get_tai_ns() - tai_offset;

    __builtin_memset(&key, 0, sizeof(key));
    __builtin_memset(&fprint, 0, sizeof(fprint));

    if (bpf_skb_load_bytes(skb, 0, &tcp, sizeof(tcp)) != 0) {
        return -1;
    }

    if (tcp.syn && !tcp.ack) {
        fprint.time = time;
        load_ttl(skb, &fprint.ttl);
        load_key(skb, &tcp, &key);
        bpf_map_update_elem(&example_fingerprint, &key, &fprint, BPF_NOEXIST);
    }

    return -1;
}
```

Fingerprinting w/ eBPF: Esempio

```
void load_key(struct __sk_buff *skb, struct tcphdr *tcp, struct tcp_fprint_key *key)
{
    if (skb->protocol == bpf_htons(ETH_P_IP))
    {
        bpf_skb_load_bytes_relative(skb, offsetof(struct iphdr, saddr),
            &key->ip, sizeof(__be32), BPF_HDR_START_NET);
    }
    else if (skb->protocol == bpf_htons(ETH_P_IPV6))
    {
        bpf_skb_load_bytes_relative(skb, offsetof(struct ipv6hdr, saddr),
            &key->ip, sizeof(key->ip), BPF_HDR_START_NET);
    }

    key->port = bpf_ntohs(tcp->source);
}
```

Fingerprinting w/ eBPF: Esempio

```
void load_ttl(struct __sk_buff *skb, unsigned char* ttl_out)
{
    if (skb->protocol == bpf_htons(ETH_P_IP))
    {
        bpf_skb_load_bytes_relative(skb, offsetof(struct iphdr, ttl),
            ttl_out, sizeof(__u8), BPF_HDR_START_NET);

    }
    else if (skb->protocol == bpf_htons(ETH_P_IPV6))
    {
        bpf_skb_load_bytes_relative(skb, offsetof(struct ipv6hdr, hop_limit),
            ttl_out, sizeof(__u8), BPF_HDR_START_NET);
    }
}
```

Domande?

Discord: **@c0mm4nd_**, **@.prosti.**