

peetch

an eBPF based networking tool

SSTIC 2023

Guillaume VALADON - @guedou



Scapy co-maintainer

Python-based packet manipulation tool

Hobbyist reverser

I like to break things to understand them

Director of Security Research @Quarkslab

we bring our cutting edge researchers to customers



What is peetch?



collections of networking tools

using eBPF for the heavy lifting

two current uses

1. sniff packets and process metadata
2. dump OpenSSL secrets



github.com/quarkslab/peetch



secdev/scapy

NSS Key Log support

PCAPNg writing & comment option

iovisor/bcc

CGROUP_SOCKET_ADDR program type support

quarkslab/peetch

the eBPF & TLS playground

eBPF 101



like cBPF but better

internally Linux convert cBPF to eBPF

cool features

designed to be JITed
function calls

attach an eBPF program to an event
packets, kernel functions...





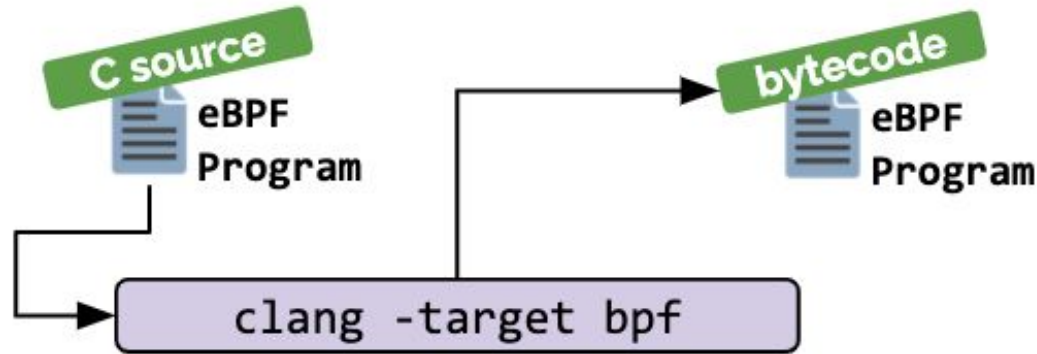
Using a kprobe Manually

```
00 # echo 'p:sstic23 do_sys_openat2 filename=+0($arg2):string' >
/sys/kernel/debug/tracing/kprobe_events
01 # echo 1 > /sys/kernel/debug/tracing/events/kprobes/sstic23/enable
02
03 # cat /sys/kernel/debug/tracing/trace_pipe
04 secret_binary-1603837 [000] d... 242641.480399: sstic23: (do_sys_openat2+0x0/0x164)
filename="/etc/ld.so.cache"
05 secret_binary-1603837 [000] d... 242641.480470: sstic23: (do_sys_openat2+0x0/0x164)
filename="/lib/aarch64-linux-gnu/libcurl.so.4"
06 secret_binary-1603837 [000] d... 242641.480594: sstic23: (do_sys_openat2+0x0/0x164)
filename="/lib/aarch64-linux-gnu/libz.so.1"
07 secret_binary-1603837 [000] dn.. 242641.480695: sstic23: (do_sys_openat2+0x0/0x164)
filename="/lib/aarch64-linux-gnu/libpthread.so.0"
```

three simple steps from a root shell

create the kprobe, enable it & enjoy

From C to eBPF bytecode



compiled to eBPF with `clang`

`gcc` can do it too

bytecode loaded with the `bpf syscall`

abstracted thanks to eBPF loaders

peetch dump

Packet process metadata with bpftrace



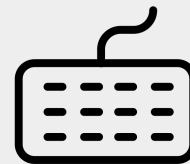
```
01 #include <net/flow.h>
02
03 kprobe:security_sk_classify_flow {
04     $flowi = (struct flowi*) arg1;
05     if ($flowi == 0) {
06         return;
07     }
08
09     $flowi4 = $flowi->u.ip4;
10     if ($flowi4.saddr == 0) {
11         return;
12     }
13
14     $sport = bswap($flowi4.uli.ports.sport);
15     $dport = bswap($flowi4.uli.ports.dport);
16
17     $saddr = ntop(AF_INET, $flowi4.saddr);
18     $daddr = ntop(AF_INET, $flowi4.daddr);
19
20     printf("%s/%d - %s:%d -> %s:%d\n", comm, pid, $saddr, $sport, $daddr, $dport);
21 }
```

bpftrace output



```
# bpftrace security_sk_classify_flow.bt
Attaching 1 probe...
secret_binary/11663 - 127.0.0.1:57338 -> 127.0.0.53:53
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 127.0.0.53:53 -> 127.0.0.1:57338
systemd-resolve/1152 - 127.0.0.53:53 -> 127.0.0.1:57338
secret_binary/11663 - 192.168.42.42:37890 -> 188.114.97.6:443
```

peetch Demo #1



Created by Sarah
from Noun Project

store process information along packets
filter TLS and display results with Scapy

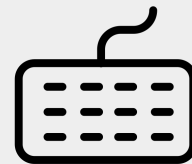
```
00 # peetch dump --write sstic23.pcapng
01 $ scapy
02 >>> rdpcap("sstic23.pcapng")
03 <sstic23.pcapng: TCP:37 UDP:0 ICMP:0 Other:0>
04 >>> l = _.filter(lambda p: TLS in p)
05 <filtered sstic23.pcapng: TCP:13 UDP:0 ICMP:0 Other:0>
06 >>> l[0].summary()
07 'Ether / IP / TCP / TLS 10.211.55.7:44154 > 208.97.177.124:443 / TLS / TLS Handshake
- Client Hello / Padding'
08 >>> l[0].comment
09 secret_binary/26380'
```

peetch t1s

peetch Demo #2

dump plaintext

HTTP/1.1 works fine!



Created by Sarah
from Noun Project

```
# peetch tls --content
<- curl (875531) 127.0.0.1/2807 TLS1.2

0000  47 45 54 20 2F 3F 73 65 63 72 65 74 3D 31 37 38  GET /?secret=178
0010  31 30 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73  10 HTTP/1.1..Hos
0020  74 3A 20 6C 6F 63 61 6C 68 6F 73 74 3A 32 38 30  t: localhost:280
0030  37 0D 0A 55 73 65 72 2D 41 67 65 6E 74 3A 20 63  7..User-Agent: c

-> curl (875531) 127.0.0.1/2807 TLS1.2

0000  48 54 54 50 2F 31 2E 30 20 32 30 30 20 6F 6B 0D  HTTP/1.0 200 ok.
0010  0A 43 6F 6E 74 65 6E 74 2D 74 79 70 65 3A 20 74  .Content-type: t
0020  65 78 74 2F 68 74 6D 6C 0D 0A 0D 0A 3C 48 54 4D  ext/html...<HTM
0030  4C 3E 3C 42 4F 44 59 20 42 47 43 4F 4C 4F 52 3D  L><BODY BGCOLOR=
```



```
<- curl (904765) 208.97.177.124/443 TLS1.3 TLS_AES_256_GCM_SHA384

0000  50 52 49 20 2A 20 48 54 54 50 2F 32 2E 30 0D 0A  PRI * HTTP/2.0..
0010  0D 0A 53 4D 0D 0A 0D 0A                               ..SM....

<- curl (904765) 208.97.177.124/443 TLS1.3 TLS_AES_256_GCM_SHA384

0000  00 00 12 04 00 00 00 00 00 00 03 00 00 00 64 00  .....d.
0010  04 40 00 00 00 00 02 00 00 00 00           .@.....

<- curl (904765) 208.97.177.124/443 TLS1.3 TLS_AES_256_GCM_SHA384

0000  00 00 04 08 00 00 00 00 00 3F FF 00 01           .....?...
```

content is much more difficult to get
must support HTTP/2 in peetch

Extracting and Exporting Keys



```
[0xfffffb0325c70]> pd 2 @ sym.SSL_get_session
;-- sym.SSL_get_session:
0xfffffb032d518      008842f9      ldr x0, [x0, 0x510]      ; [0x510:4]=-1 ; 1296
0xfffffb032d51c      c0035fd6      ret
```

extract keys from the `ssl_session_st` structure

offsets could be guessed at runtime

export keys in a NSS Key Log

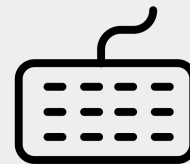
simple plaintext format

supports TLS1.2 & TLS 1.3

tools support: Firefox, curl, Wireshark, Scapy...

peetch Demo #3

decrypting secret_binary TLS traffic



Created by Sarah
from Noun Project

```
01 $ (sleep 5; secret_binary) &
02 # peetch tls --write &
secret_binary (930462) 188.114.96.6/443 TLS1.2 ECDHE-ECDSA-CHACHA20-POLY1305
03 # peetch dump --write traffic.pcapng
04 ^C
05
06 $ editcap --inject-secrets tls,930462-master_secret.log traffic.pcapng traffic-ms.pcapng
07
08 $ scapy
09 >>> load_layer("tls")
10 >>> conf.tls_session_enable = True
11 >>> l = rdpcap("traffic-ms.pcapng")
12 >>> l[13][TLS].msg
[<TLSApplicationData data='GET /?name=highly%20secret%20information HTTP/1.1\r\nHost:
hello.guedou.workers.dev\r\nUser-Agent: curl/7.68.0\r\nAccept: */*\r\n\r\n' |>]
```

WIP - Transparent TLS Proxy



```
01 # peetch proxy
02 [!] Proxying OpenSSL traffic
03 [+] Decrypting traffic to 208.97.177.124/443 via 127.0.0.1/33305
04 <-- 10.211.55.10 > 208.97.177.124 tcp
05 --> 208.97.177.124 > 127.0.0.1 tcp
06 --> 208.97.177.124 > 127.0.0.1 tcp
07 --> 208.97.177.124 > 127.0.0.1 tcp
08 <-- 10.211.55.10 > 208.97.177.124 tcp
09 --> 208.97.177.124 > 127.0.0.1 tcp
10 <-- 10.211.55.10 > 208.97.177.124 tcp
11 --> 208.97.177.124 > 127.0.0.1 tcp
12 <-- 10.211.55.10 > 208.97.177.124 tcp
13
14 ###[ TLS Application Data ]###
15 data      = 'GET /?name=highly%20secret%20information HTTP/1.1\r\nHost: www.perdu.com\r\nUser-Agent:
curl/7.68.0\r\nAccept: */*\r\n\r\n'
```

Questions?
Issues?
PR?





experiment with eBPF as a security tool

try it yourself back home

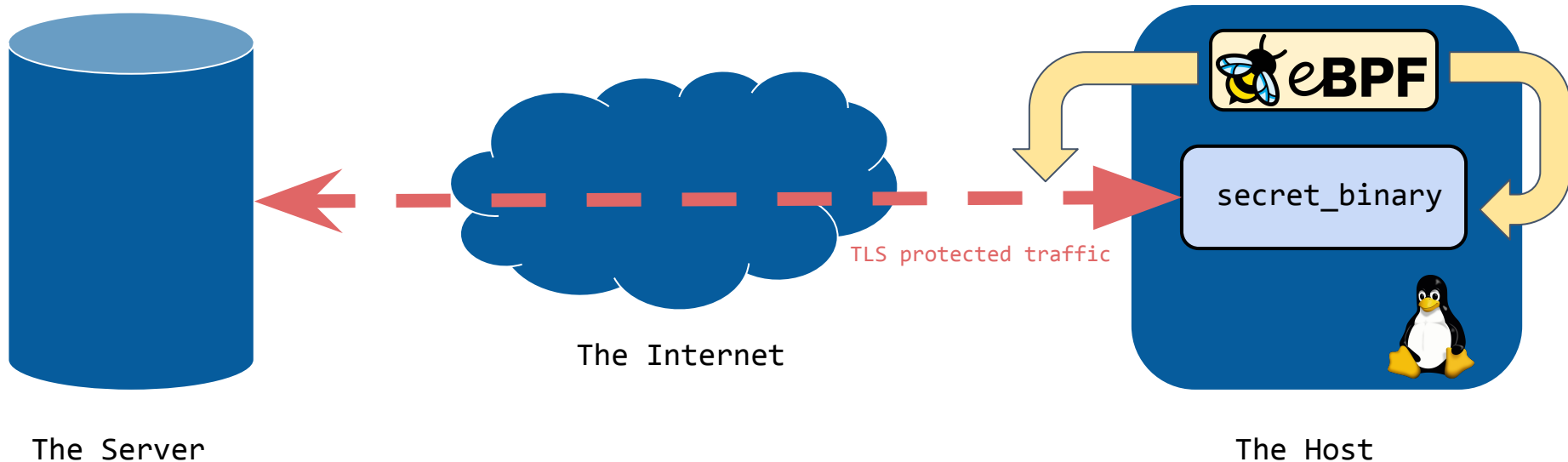
discuss uses-cases with examples

focus on TLS analysis

snoop on a local process

network traffic, function calls...

Today Setup



eBPF used to snoop on the secret_binary process

it runs on the host as root



identify a **local process**
network activity on Linux



save a network capture and associated processes

capture now and analyze later

discard traffic associated with a noisy process

SSH and Firefox are usually not interesting

focus attention on a specific process

malware only traffic



alternatives yet missing process information

two candidate solutions

1. **sniff a specific user traffic**
using iptables NFLOG feature
2. **sniff a specific process traffic**
using a dedicated network namespace



get **cryptographic keys** from
memory and decrypt traffic



defeat certificate pinning

no need to patch the binary

decrypt with networking tools

like Wireshark, Scapy



succession of unauthenticated messages

ClientHello, ServerHello, Certificate...

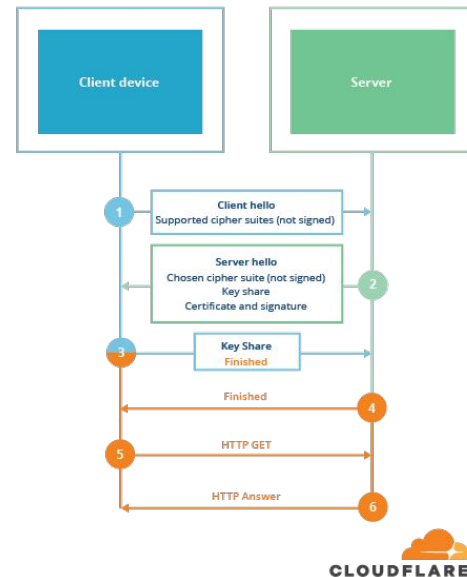
trusting the server is enough

must validate its certificate

resulting **shared cryptographic session keys**

used to protect traffic

TLS 1.2 ECDHE





steal server RSA private key

nowadays not pertinent

does not defeat Perfect Forward Secrecy

access TLS session keys

get the master secret



- 1 . explore OpenSSL structures
not designed to be accessed externally
- 2 . get a pointer to the master secret
from `SSL_get_session()`
- 3 . get the TLS ciphersuite
enhance the output

struct ssl_session_st pointer

```
[0xffffb0325c70]> pd 2 @ sym.SSL_get_session
;-- sym.SSL_get_session:
0xffffb032d518      008842f9      ldr x0, [x0, 0x510]      ; [0x510:4]=-1 ; 1296
0xffffb032d51c      c0035fd6      ret
```

offset to the `ssl_session_st` structure

could be emulated with miasm, or radare2 ESIL



NSS Key Log

simple plaintext format

supports TLS1.2 & TLS 1.3

many tools supports it

exporters: Firefox, curl...

importers: Wireshark, Scapy...

Extract Master Secret from ssl_session_st

```
# peetch tls --secrets  
<- secret_binary (907675) 188.114.97.6/2807 TLS1.2 ECDHE-RSA-AES256-GCM-SHA384  
CLIENT_RANDOM 28071980 c3266a14409a49fc482a0d9074 .. 2830fccce7325b1e69e73dd4a28ee79a3eceb
```

unsigned char master_key[TLS13_MAX_RESUMPTION_PSK_LENGTH];
it is 48 bytes for TLS1.2

Decrypting secret_binary TLS Traffic

```
01 $ (sleep 5; secret_binary) &
02 # peetch tls --write &
secret_binary (930462) 188.114.96.6/443 TLS1.2 ECDHE-ECDSA-CHACHA20-POLY1305
03 # peetch dump --write traffic.pcapng
04 ^C
05
06 $ editcap --inject-secrets tls,930462-master_secret.log traffic.pcapng traffic-ms.pcapng
07
08 $ scapy
09 >>> load_layer("tls")
10 >>> conf.tls_session_enable = True
11 >>> l = rdpcap("traffic-ms.pcapng")
12 >>> l[13][TLS].msg
[<TLSApplicationData data='GET /?name=highly%20secret%20information HTTP/1.1\r\nHost:
hello.guedou.workers.dev\r\nUser-Agent: curl/7.68.0\r\nAccept: */*\r\n\r\n' |>]
```



secdev/scapy

NSS Key Log support

PCAPNg writing & comment option

iovisor/bcc

CGROUP_SOCKET_ADDR program type support

quarkslab/peetch

the eBPF & TLS playground

Get the slides at



eBPF 101



define network packets filters

commonly used with `tcpdump`

filter expressions compiled to BPF bytecode

host 1.1.1.1 and tcp and port 443

attached to a RAW socket and processed by the kernel

`SO_ATTACH_FILTER` or `SO_ATTACH_BPF`

Convert a Filter to Bytecode

```
# tcpdump -i eth0 -d 'host 1.1.1.1 and tcp and port 443'
(000) ldh      [12]
(001) jeq      #0x800          jt 2      jf 16
(002) ld       [26]
(003) jeq      #0x1010101     jt 6      jf 4
(004) ld       [30]
(005) jeq      #0x1010101     jt 6      jf 16
(006) ldb      [23]
(007) jeq      #0x6           jt 8      jf 16
(008) ldh      [20]
(009) jset     #0x1fff        jt 16     jf 10
(010) ldx     4*([14]&0xf)
(011) ldh      [x + 14]
(012) jeq      #0x1bb         jt 15     jf 13
(013) ldh      [x + 16]
(014) jeq      #0x1bb         jt 15     jf 16
(015) ret      #262144
(016) ret      #0
```

Using a filter

```
# tcpdump -ni eth0 -X 'host 1.1.1.1 and tcp and port 443'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
17:34:44.647471 IP 10.211.55.4.39712 > 1.1.1.1.443: Flags [S], seq 649388654, win 64240,
options [mss 1460,sackOK,TS val 951387719 ecr 0,nop,wscale 7], length 0
    0x0000:  4500 003c 360c 4000 4006 c0d7 0ad3 3704  E..<6.@.@.....7.
    0x0010:  0101 0101 9b20 01bb 26b4 e26e 0000 0000  .....&..n....
    0x0020:  a002 faf0 4407 0000 0204 05b4 0402 080a  ....D.....
    0x0030:  38b5 0647 0000 0000 0103 0307                8..G.....
17:34:44.676148 IP 1.1.1.1.443 > 10.211.55.4.39712: Flags [S.], seq 1336932985, ack 649388655,
win 32768, options [mss 1460,wscale 1,nop], length 0
    0x0000:  4500 0030 ed3c 0000 8006 09b3 0101 0101  E..0.<.....
    0x0010:  0ad3 3704 01bb 9b20 4faf fa79 26b4 e26f  ..7.....0..y&..o
    0x0020:  7012 8000 d00c 0000 0204 05b4 0303 0101  p.....
```



like cBPF but better

internally Linux convert cBPF to eBPF

cool features

designed to be JITed
function calls

attach an eBPF program to an event
packets, kernel functions...





eBPF programs types

XDP, TRACEPOINT, UPROBE, KPROBE...

eBPF maps

share data between kernel and userspace

eBPF verifier

ensure programs safety
your worst best friend



break into any kernel function

INT3 on i386 and x86_64

collect information

can't modify functions behavior

two types

kprobe - anywhere

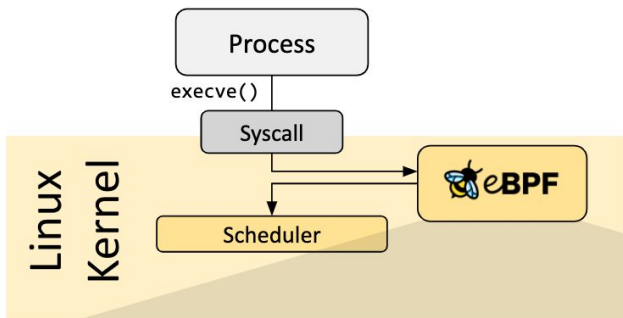
kretprobe - when a function returns

Using a kprobe Manually

```
00 # echo 'p:sstic23 do_sys_openat2 filename=+0($arg2):string' >
/sys/kernel/debug/tracing/kprobe_events
01 # echo 1 > /sys/kernel/debug/tracing/events/kprobes/sstic23/enable
02
03 # cat /sys/kernel/debug/tracing/trace_pipe
04 secret_binary-1603837 [000] d... 242641.480399: sstic23: (do_sys_openat2+0x0/0x164)
filename="/etc/ld.so.cache"
05 secret_binary-1603837 [000] d... 242641.480470: sstic23: (do_sys_openat2+0x0/0x164)
filename="/lib/aarch64-linux-gnu/libcurl.so.4"
06 secret_binary-1603837 [000] d... 242641.480594: sstic23: (do_sys_openat2+0x0/0x164)
filename="/lib/aarch64-linux-gnu/libz.so.1"
07 secret_binary-1603837 [000] dn.. 242641.480695: sstic23: (do_sys_openat2+0x0/0x164)
filename="/lib/aarch64-linux-gnu/libpthread.so.0"
```

three simple steps from a root shell
create the kprobe, enable it & enjoy

Process Execution Example



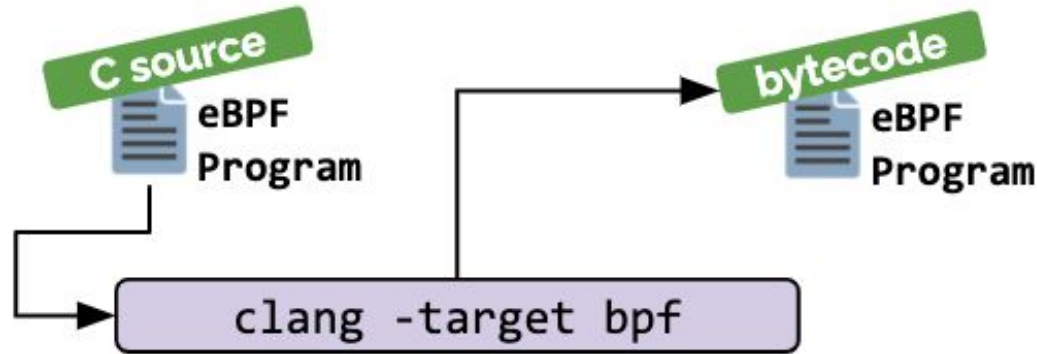
```
int syscall__ret_execve(struct pt_regs *ctx)
{
    struct comm_event event = {
        .pid = bpf_get_current_pid_tgid() >> 32,
        .type = TYPE_RETURN,
    };

    bpf_get_current_comm(&event.comm, sizeof(event.comm));
    comm_events.perf_submit(ctx, &event, sizeof(event));

    return 0;
}
```

1. attach to execve syscall
2. get PID and name
3. submit them to a map

From C to eBPF bytecode



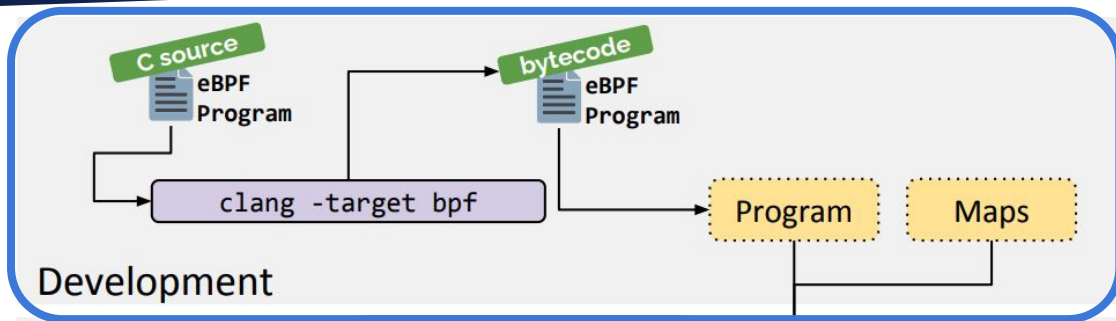
compiled to eBPF with **clang**

gcc can do it too

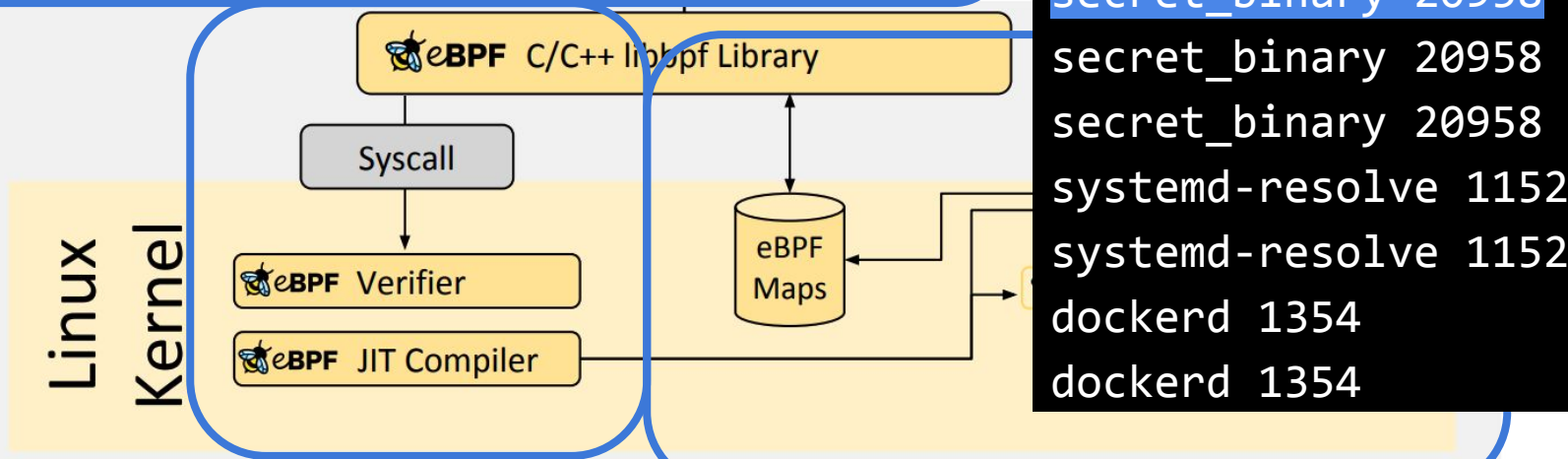
bytecode loaded with the **bpf syscall**

abstracted thanks to eBPF loaders

The High-Level Overview



Development



Runtime

```
dockerd 1354
systemd-resolve 1152
systemd 1
secret_binary 20958
secret_binary 20958
secret_binary 20958
systemd-resolve 1152
systemd-resolve 1152
dockerd 1354
dockerd 1354
```



limited languages options

C, Rust, awk-like

many loaders

Go, Rust, Python, bpftrace

beloved security tools works!

Ghidra, objdump, readelf...

eBPF & Networking



identify a **local process**
network activity on Linux



save a network capture and associated processes

capture now and analyze later

discard traffic associated with a noisy process

SSH and Firefox are usually not interesting

focus attention on a specific process

malware only traffic



alternatives yet missing process information

two candidate solutions

1. **sniff a specific user traffic**
using iptables NFLOG feature
2. **sniff a specific process traffic**
using a dedicated network namespace

Existing Tools on macOS

```
# tcpdump -i pktap,en0 host 1.1.1.1 -w out.pcapng
tcpdump: data link type PKTAP
tcpdump: listening on pktap,en0, link-type PKTAP (Apple DLT_PKTAP), capture
size 262144 bytes
2 packets captured
78 packets received by filter
0 packets dropped by kernel
```

macOS tcpdump do it natively

specify `pktap` as the interface name

Existing Tools on macOS

```
$ tcpdump -k -r out.pcapng
reading from PCAP-NG file out.pcapng
14:31:36.627265 (en0, proc dig:10855, svc BE, out, so) IP 10.0.3.61.57044 >
one.one.one.one.domain: 26562+ [1au] A? www.example.org. (44)
14:31:36.640010 (en0, proc dig:10855, svc BE, in, so) IP
one.one.one.one.domain > 10.0.3.61.57044: 26562$ 1/0/1 A 93.184.216.34 (60)
```

process information stored in PCAPng

Apple specific option understood by Wireshark

Existing Tools on Windows

Administrator: Windows PowerShell

```
PS C:\> netsh trace start capture=yes tracefile=c:\tests\capture.etl persistent=yes
```

Trace configuration:

```
-----  
Status:          Running  
Trace File:      c:\tests\capture.etl  
Append:         Off  
Circular:        On  
Max Size:       512 MB  
Report:         Off
```

```
PS C:\> netsh trace stop  
Merging traces ... done  
Generating data collection ... done  
The trace file and additional troubleshooting information have been compiled as "c:\tests\capture.cab".  
File location = c:\tests\capture.etl  
Tracing session was successfully stopped.
```

```
PS C:\>
```

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.
Try the new cross-platform PowerShell <https://aka.ms/powershell>

```
PS C:\Users\admin> ping 8.8.8.8
```

Pinging 8.8.8.8 with 32 bytes of data:

```
Reply from 8.8.8.8: bytes=32 time=6ms TTL=115  
Reply from 8.8.8.8: bytes=32 time=9ms TTL=115  
Reply from 8.8.8.8: bytes=32 time=7ms TTL=115  
Reply from 8.8.8.8: bytes=32 time=8ms TTL=115
```

Ping statistics for 8.8.8.8:

```
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),  
Approximate round trip times in milli-seconds:  
  Minimum = 6ms, Maximum = 9ms, Average = 7ms
```

```
PS C:\Users\admin> curl.exe http://wttr.in/Paris
```

Weather report: Paris

Sunny
19 °C
26 km/h
10 km
0.0 mm

Thu 28 Apr			
Morning	Noon	Evening	Night
Partly cloudy +14(13) °C 14-16 km/h 10 km 0.0 mm 0%	Partly cloudy 19 °C 18-21 km/h 10 km 0.0 mm 0%	Cloudy 20 °C 15-19 km/h 10 km 0.0 mm 0%	Sunny 16 °C 8-15 km/h 10 km 0.0 mm 0%

Fri 29 Apr			
Morning	Noon	Evening	Night
Overcast +13(12) °C 10-13 km/h 10 km 0.0 mm 0%	Partly cloudy 15 °C 16-18 km/h 10 km 0.0 mm 0%	Cloudy 16 °C 14-19 km/h 10 km 0.0 mm 0%	Sunny +14(13) °C 12-16 km/h 10 km 0.0 mm 0%

netsh trace
capture=yes

trace saved in ETL

Existing Tools on Windows

The screenshot shows two windows. The top window is an Administrator Windows PowerShell terminal with the following commands and output:

```
PS C:\tests> curl.exe -LOf https://github.com/microsoft/etl2pcapng/releases/download/v1.7.0/etl2pcapng.zip
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 653 100 653 0 0 1444 0 ---:--:--:--:--:--:--:--:--: 1444
100 1369k 100 1369k 0 0 1620k 0 ---:--:--:--:--:--:--:--:--: 1620k
PS C:\tests> .\etl2pcapng\etl2pcapng\x64\etl2pcapng.exe
etl2pcapng <infile> <outfile>
Converts a packet capture from etl to pcapng format.
PS C:\tests> .\etl2pcapng\etl2pcapng\x64\etl2pcapng.exe .\capture.etl .\capture.pcapng
IP: mediumeth ID=0 IfIndex=13 VlanID=0
Converted 46 frames
PS C:\tests>
```

The bottom window is Wireshark displaying the converted capture file 'capture.pcapng'. The packet list pane shows the following data:

No.	Time	Source	Destination	Protocol	Length	Info
22	52.916819	192.168.1.254	192.168.48.25	DNS	83	Standard query response 0xb0a0 A wtrtr.in A 5.9.243.187
23	52.920752	192.168.48.25	5.9.243.187	TCP	66	50144 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
24	52.941845	5.9.243.187	192.168.48.25	TCP	60	80 → 50144 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0 MSS=1460
25	52.941938	192.168.48.25	5.9.243.187	TCP	54	50144 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
26	52.948864	192.168.48.25	5.9.243.187	HTTP	130	GET /Paris HTTP/1.1
27	53.041504	5.9.243.187	192.168.48.25	TCP	60	80 → 50144 [ACK] Seq=1 Ack=77 Win=32692 Len=0
28	57.932281	51.105.236.244	192.168.48.25	TCP	60	443 → 50139 [FIN, ACK] Seq=1 Ack=1 Win=31725 Len=0

The packet details pane shows the selected packet (No. 26) with the following structure:

- Packet comments
 - PID=5772
 - [Expert Info (Comment/Comment): PID=5772]
 - [PID=5772]
 - [Severity level: Comment]
 - [Group: Comment]
- Ethernet II, Src: PcsCompu_69:00:8a (08:00:27:69:00:8a), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
- Internet Protocol Version 4, Src: 192.168.48.25, Dst: 5.9.243.187
- Hypertext Transfer Protocol

The packet bytes pane shows the raw hex and ASCII data for the selected packet.

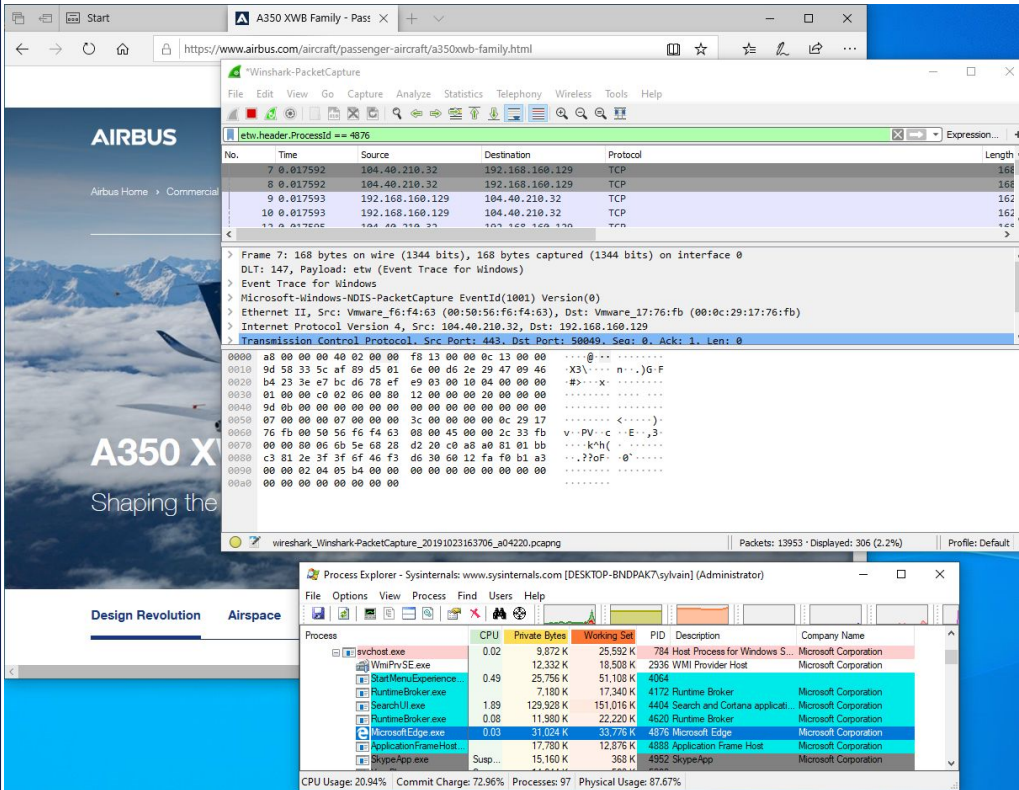
etl2pcapng tool

convert ETL to pcapng

visualize in Wireshark

info in PCAPng comments

Existing Tools - Windows



Winshark captures ETW
mix system and network information



1 . sniff packets

with eBPF

2 . get per packet process information

process name and ID

3 . identify TLS messages

with Scapy



custom Traffic Control (TC)

an eBPF program can alter traffic

two custom programs exist

custom classifier CLS BPF_PROG_TYPE_SCHED_CLS

custom actions ACT BPF_PROG_TYPE_SCHED_ACT

how?

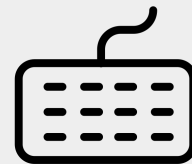
access packets in a TC classifier

send them to userland using an eBPF map

Simple eBPF TC Classifier in C

```
01 int process_frame(struct __sk_buff *skb)
02 {
03     // Data accessors
04     unsigned char *data = (void *)(long)skb->data;
05     unsigned char *data_end = (void *)(long)skb->data_end;
06
07     // Mapping data to the Ethernet and IP headers
08     struct ethhdr *eth = (struct ethhdr *)data;
09     struct iphdr *iph = (struct iphdr*) (data + sizeof(struct ethhdr));
10
11     // Simple length check
12     if ((data + sizeof(struct ethhdr) + sizeof(struct iphdr)) > data_end)
13         return TC_ACT_OK;
14
15     // Discard everything but IPv4 and TCP
16     if (ntohs(eth->h_proto) != ETH_P_IP && iph->protocol != IPPROTO_TCP)
17         return TC_ACT_OK;
18
19     bool _unused = true;
20     skb_events.perf_submit_skb(skb, skb->len, &_unused, sizeof(_unused));
21
22     return TC_ACT_OK;
23 }
```

Peetch Demo #1



Created by Sarah
from Noun Project

sniff packets with eBPF
parsed and displayed with Scapy

```
# peetch dump --raw
Ether / IP / TCP 10.211.55.7:43908 > 208.97.177.124:https S / Padding
Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43908 SA / Padding
Ether / IP / TCP 10.211.55.7:43908 > 208.97.177.124:https A / Padding
Ether / IP / TCP 10.211.55.7:43908 > 208.97.177.124:https PA / Raw / Padding
Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43908 A / Padding
Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43908 PA / Raw / Padding
Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43908 PA / Raw / Padding
Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43908 PA / Raw / Padding
Ether / IP / TCP 10.211.55.7:43908 > 208.97.177.124:https A / Padding
```



`bpf_get_current_pid_tgid()` helper not available
cannot use it in BPF_PROG_TYPE_SCHED_ACT programs

`struct flowi` describes packets parameters
used for IPsec SA lookups
accessible in `stable LSM hooks`

Per Packet PID Lookup Strategy



1 . find a hook point

for example `security_sk_classify_flow()`

2 . access the PID

using the eBPF helper function

3 . write the PID into an eBPF map

use IP & ports as indexes

security_sk_classify_flow() kprobe with bpftrace

```
01 #include <net/flow.h>
02
03 kprobe:security_sk_classify_flow {
04     $flowi = (struct flowi*) arg1;
05     if ($flowi == 0) {
06         return;
07     }
08
09     $flowi4 = $flowi->u.ip4;
10     if ($flowi4.saddr == 0) {
11         return;
12     }
13
14     $sport = bswap($flowi4.uli.ports.sport);
15     $dport = bswap($flowi4.uli.ports.dport);
16
17     $saddr = ntop(AF_INET, $flowi4.saddr);
18     $daddr = ntop(AF_INET, $flowi4.daddr);
19
20     printf("%s/%d - %s:%d -> %s:%d\n", comm, pid, $saddr, $sport, $daddr, $dport);
21 }
```

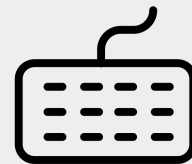
security_sk_classify_flow() kprobe Output

```
# bpftrace security_sk_classify_flow.bt
Attaching 1 probe...
secret_binary/11663 - 127.0.0.1:57338 -> 127.0.0.53:53
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 42.1.203.8:27903 -> 12.51.197.0:47654
systemd-resolve/1152 - 127.0.0.53:53 -> 127.0.0.1:57338
systemd-resolve/1152 - 127.0.0.53:53 -> 127.0.0.1:57338
secret_binary/11663 - 192.168.42.42:37890 -> 188.114.97.6:443
```




- 1 . compute a packet hash
from IP & ports
- 2 . use hash to lookup PID information
match PID & packets
- 3 . display everything
relax

Peetch Demo #2



Created by Sarah
from Noun Project

display process information along packets
retrieved using an eBPF map

```
# peetch dump
curl/24102 - Ether / IP / TCP 10.211.55.7:43912 > 208.97.177.124:https S / Padding
curl/24102 - Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43912 SA / Padding
curl/24102 - Ether / IP / TCP 10.211.55.7:43912 > 208.97.177.124:https A / Padding
curl/24102 - Ether / IP / TCP 10.211.55.7:43912 > 208.97.177.124:https PA / Raw / Padding
curl/24102 - Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43912 A / Padding
curl/24102 - Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43912 PA / Raw / Padding
curl/24102 - Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43912 PA / Raw / Padding
curl/24102 - Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43912 PA / Raw / Padding
curl/24102 - Ether / IP / TCP 208.97.177.124:https > 10.211.55.7:43912 PA / Raw / Padding
curl/24102 - Ether / IP / TCP 10.211.55.7:43912 > 208.97.177.124:https A / Padding
```



PCAPng is a good candidate

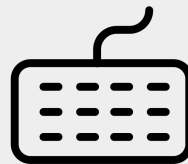
- extensible packet capture format
- options can be added to packet

comment option

- can store any text
- supported by Wireshark, tcpdump, Scapy...

Peetch Demo #3

store process information along packets
filter TLS and display results with Scapy



Created by Sarah
from Noun Project

```
00 # peetch dump --write csw22.pcapng
01 $ scapy
02 >>> rdpcap("csw22.pcapng")
03 <csw22.pcapng: TCP:37 UDP:0 ICMP:0 Other:0>
04 >>> l = _.filter(lambda p: TLS in p)
05 <filtered test.pcapng: TCP:13 UDP:0 ICMP:0 Other:0>
06 >>> l[0].summary()
07 'Ether / IP / TCP / TLS 10.211.55.7:44154 > 208.97.177.124:443 / TLS / TLS Handshake
- Client Hello / Padding'
08 >>> l[0].comment
09 secret_binary/26380'
```

eBPF & Userspace



identify when a process uses
OpenSSL functions



observe a process behavior

hook functions and execute eBPF program

access data manipulated by OpenSSL

plaintext is nearby!

Setting an OpenSSL uprobe Manually

```
# echo "p:SSL_read /lib/aarch64-linux-gnu/libssl.so.1.1:0x34990" >
/sys/kernel/debug/tracing/uprobe_events

# echo 1 > /sys/kernel/debug/tracing/events/uprobes/SSL_read/enable

# cat /sys/kernel/debug/tracing/trace_pipe
secret_binary-72453 [000] ... 33213.115372: SSL_read: (0xffffab025990)
secret_binary-72453 [000] ... 33213.116449: SSL_read: (0xffffab025990)
secret_binary-72453 [000] ... 33213.126230: SSL_read: (0xffffab025990)
secret_binary-72453 [000] ... 33213.126467: SSL_read: (0xffffab025990)
```

three simple steps from a root shell
create the uprobe, enable it & enjoy

Setting an OpenSSL uprobe with bpftrace

```
# bpftrace -e 'uprobe:/lib/aarch64-linux-gnu/libssl.so:SSL_read
{ printf("SSL_read() from %s (%d)\n", comm, pid); }'
Attaching 1 probe...
SSL_read() from secret_binary (306772)
SSL_read() from secret_binary (306772)
SSL_read() from secret_binary (306796)
```

hook and display information with a one-liner
perfect way to experiment with eBPF



1 . hook connect()

send destination IP and port to an eBPF map

2 . hook TLS libraries functions

like SSL_read, or gnutls_record_recv

3 . mix information inside TLS hooks

retrieve data from connect()

send an event to userland

connect() tracepoint with bpftrace

```
01 #include <linux/in.h>
02
03 tracepoint:syscalls:sys_enter_connect {
04     $addr_in = (struct sockaddr_in *)args->uservaddr;
05
06     if ($addr_in->sin_family == 2) { // AF_INET
07         $addr = ntop($addr_in->sin_family, $addr_in->sin_addr.s_addr);
08
09         if ($addr_in->sin_port > 0) {
10             printf("%s/%d\n", $addr, bswap($addr_in->sin_port));
11         }
12     }
13 }
```

access sockaddr_in structure

check values and print information

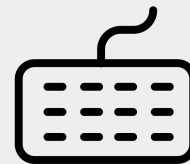
connect() tracepoint Output

```
# bpftrace sys_enter_connect.bt
Attaching 1 probe...
127.0.0.53/53
188.114.97.6/443
```

monitor on-going TCP connections

create the uprobe, enable it & enjoy

Peetch Demo #3



Created by Sarah
from Noun Project

show processes using OpenSSL
get IP and ports from connect()

```
# peetch tls --directions
<- secret_binary (61673) 188.114.97.6/443
-> secret_binary (61673) 188.114.97.6/443
-> secret_binary (61673) 188.114.97.6/443
<- mutt (16453) 85.65.32.2/993
-> mutt (16453) 85.65.32.2/993
-> mutt (16453) 85.65.32.2/993
```

Accessing Unencrypted Data

SSL_read

NAME

SSL_read_ex, SSL_read, SSL_peek_ex, SSL_peek - read bytes from a TLS/SSL connection

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_read_ex(SSL *ssl, void *buf, size_t num, size_t *readbytes);
int SSL_read(SSL *ssl, void *buf, int num);

int SSL_peek_ex(SSL *ssl, void *buf, size_t num, size_t *readbytes);
int SSL_peek(SSL *ssl, void *buf, int num);
```

DESCRIPTION

SSL_read_ex() and SSL_read() try to read **num** bytes from the specified **ssl** into the buffer **buf**. On success SSL_read_ex() will store the number of bytes actually read in ***readbytes**.

SSL_peek_ex() and SSL_peek() are identical to SSL_read_ex() and SSL_read() respectively except no bytes are actually removed from the underlying BIO during the read, so that a subsequent call to SSL_read_ex() or SSL_read() will yield at least the same bytes.

NOTES

In the paragraphs below a "read function" is defined as one of SSL_read_ex(), SSL_read(), SSL_peek_ex() or SSL_peek().

If necessary, a read function will negotiate a TLS/SSL session, if not already explicitly performed by [SSL_connect\(3\)](#) or [SSL_accept\(3\)](#). If the peer requests a re-negotiation, it will be performed transparently during the read function operation. The behaviour of the read functions depends on the underlying BIO.

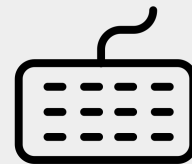
OpenSSL manipulates plaintext
eBPF programs can access it

a uretprobe is needed for SSL_read
data is only available after its call

Peetch Demo #4

dump plaintext

HTTP/1.1 works fine!



Created by Sarah
from Noun Project

```
# peetch tls --content
<- curl (875531) 127.0.0.1/2807 TLS1.2

0000  47 45 54 20 2F 3F 73 65 63 72 65 74 3D 31 37 38  GET /?secret=178
0010  31 30 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73  10 HTTP/1.1..Hos
0020  74 3A 20 6C 6F 63 61 6C 68 6F 73 74 3A 32 38 30  t: localhost:280
0030  37 0D 0A 55 73 65 72 2D 41 67 65 6E 74 3A 20 63  7..User-Agent: c

-> curl (875531) 127.0.0.1/2807 TLS1.2

0000  48 54 54 50 2F 31 2E 30 20 32 30 30 20 6F 6B 0D  HTTP/1.0 200 ok.
0010  0A 43 6F 6E 74 65 6E 74 2D 74 79 70 65 3A 20 74  .Content-type: t
0020  65 78 74 2F 68 74 6D 6C 0D 0A 0D 0A 3C 48 54 4D  ext/html...<HTM
0030  4C 3E 3C 42 4F 44 59 20 42 47 43 4F 4C 4F 52 3D  L><BODY BGCOLOR=
```



HTTP/2 is now common

it is a binary protocol

HTTP/1.1 answers are usually compressed

HTTP/2

```
<- curl (904765) 208.97.177.124/443 TLS1.3 TLS_AES_256_GCM_SHA384

0000  50 52 49 20 2A 20 48 54 54 50 2F 32 2E 30 0D 0A  PRI * HTTP/2.0..
0010  0D 0A 53 4D 0D 0A 0D 0A                                ..SM....

<- curl (904765) 208.97.177.124/443 TLS1.3 TLS_AES_256_GCM_SHA384

0000  00 00 12 04 00 00 00 00 00 00 03 00 00 00 64 00  .....d.
0010  04 40 00 00 00 00 02 00 00 00 00                                .@.....

<- curl (904765) 208.97.177.124/443 TLS1.3 TLS_AES_256_GCM_SHA384

0000  00 00 04 08 00 00 00 00 00 00 3F FF 00 01  .....?...
```

content is much more difficult to get
must support HTTP/2 in peetch

Dump TLS Keys and Decrypt Traffic



get **cryptographic keys** from
memory and decrypt traffic



defeat certificate pinning

no need to patch the binary

decrypt with networking tools

like Wireshark, Scapy



succession of unauthenticated messages

ClientHello, ServerHello, Certificate...

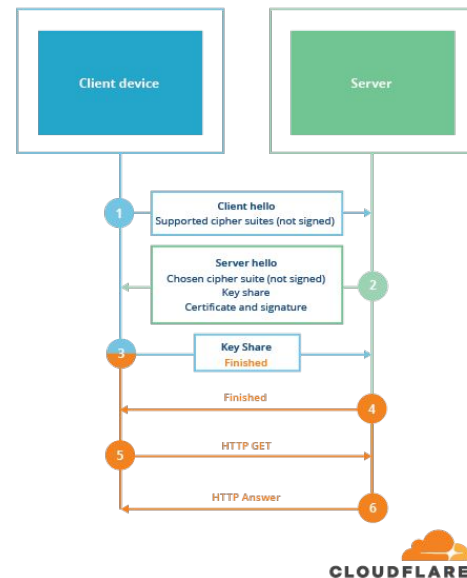
trusting the server is enough

must validate its certificate

resulting **shared cryptographic session keys**

used to protect traffic

TLS 1.2 ECDHE





steal server RSA private key

nowadays not pertinent

does not defeat Perfect Forward Secrecy

access TLS session keys

get the master secret



- 1 . explore OpenSSL structures
not designed to be accessed externally
- 2 . get a pointer to the master secret
from `SSL_get_session()`
- 3 . get the TLS ciphersuite
enhance the output

struct ssl_session_st pointer

```
[0xffffb0325c70]> pd 2 @ sym.SSL_get_session
;-- sym.SSL_get_session:
0xffffb032d518      008842f9      ldr x0, [x0, 0x510]      ; [0x510:4]=-1 ; 1296
0xffffb032d51c      c0035fd6      ret
```

offset to the `ssl_session_st` structure

could be emulated with miasm, or radare2 ESIL

Extract the TLS Cipher Suite

```
# peetch tls --content
<- curl (894762) 127.0.0.1/2807 TLS1.3 TLS_AES_256_GCM_SHA384

0000 47 45 54 20 2F 3F 73 65 63 72 65 74 3D 31 35 34 GET /?secret=154
0010 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3 HTTP/1.1..Host
0020 3A 20 6C 6F 63 61 6C 68 6F 73 74 3A 32 38 30 37 : localhost:2807
0030 0D 0A 55 73 65 72 2D 41 67 65 6E 74 3A 20 63 75 ..User-Agent: cu

-> curl (894762) 127.0.0.1/2807 TLS1.3 TLS_AES_256_GCM_SHA384

0000 48 54 54 50 2F 31 2E 30 20 32 30 30 20 6F 6B 0D HTTP/1.0 200 ok.
0010 0A 43 6F 6E 74 65 6E 74 2D 74 79 70 65 3A 20 74 .Content-type: t
0020 65 78 74 2F 70 6C 61 69 6E 0D 0A 0D 0A 45 72 72 ext/plain...Err
0030 6F 72 20 6F 70 65 6E 69 6E 67 20 27 3F 73 65 63 or opening '?sec
```

struct ssl_cipher_t *cipher;
that's a string!



NSS Key Log

simple plaintext format

supports TLS1.2 & TLS 1.3

many tools supports it

exporters: Firefox, curl...

importers: Wireshark, Scapy...

Extract Master Secret from ssl_session_st

```
# peetch tls --secrets
<- secret_binary (907675) 188.114.97.6/2807 TLS1.2 ECDHE-RSA-AES256-GCM-SHA384
CLIENT_RANDOM 28071980 c3266a14409a49fc482a0d9074 .. 2830fccce7325b1e69e73dd4a28ee79a3eceb
```

unsigned char master_key[TLS13_MAX_RESUMPTION_PSK_LENGTH];
it is 48 bytes for TLS1.2

Decrypting secret_binary TLS Traffic

```
01 $ (sleep 5; secret_binary) &
02 # peetch tls --write &
secret_binary (930462) 188.114.96.6/443 TLS1.2 ECDHE-ECDSA-CHACHA20-POLY1305
03 # peetch dump --write traffic.pcapng
04 ^C
05
06 $ editcap --inject-secrets tls,930462-master_secret.log traffic.pcapng traffic-ms.pcapng
07
08 $ scapy
09 >>> load_layer("tls")
10 >>> conf.tls_session_enable = True
11 >>> l = rdpcap("traffic-ms.pcapng")
12 >>> l[13][TLS].msg
[<TLSApplicationData data='GET /?name=highly%20secret%20information HTTP/1.1\r\nHost:
hello.guedou.workers.dev\r\nUser-Agent: curl/7.68.0\r\nAccept: */*\r\n\r\n' |>]
```

Transparent Traffic Interception



intercept OpenSSL traffic
transparently and decrypt it
on the fly



demonstrate elaborated eBPF usage

many programs and maps interacting

simplify TLS traffic manipulation

plug it to burp, mitmproxy...



alter socket behaviors

connect(), bind()...

change connect() destination IP & port

could also change source IP & port

Modify connect() Behavior with eBPF

```
00 int connect_v4_prog(struct bpf_sock_addr *ctx)
01 {
02     if (bpf_ntohs(ctx->user_port) == 53) {
03         bpf_trace_printk("connect_v4_prog() - %d\\n", bpf_ntohs(ctx->user_port));
04         ctx->user_ip4 = bpf_htonl(0x01010101); // 1.1.1.1
05     }
06     return 1;
07 }
```

redirect all TCP DNS traffic
to 1.1.1.1

Boring dig Output

```
00 $ dig @8.8.8.8 +tcp +nsid www.perdu.com
01
02 ; <<>> DiG 9.16.8-Ubuntu <<>> @8.8.8.8 +tcp +nsid www.perdu.com
03 ;; global options: +cmd
04 ;; Got answer:
05 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24528
06 ;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
07
08 ;; OPT PSEUDOSECTION:
09 ; EDNS: version: 0, flags:; udp: 1232
10 ; NSID: 31 39 6d 34 32 39 ("19m429")
11 ;; QUESTION SECTION:
12 ;www.perdu.com.                IN      A
13
14 ;; ANSWER SECTION:
15 www.perdu.com.                4198    IN      A      208.97.177.124
16
17 ;; Query time: 11 msec
18 ;; SERVER: 8.8.8.8#53(8.8.8.8)
19 ;; WHEN: Wed May 18 12:15:10 PDT 2022
20 ;; MSG SIZE rcvd: 68
```

8.8.8.8 does not
support NSID
1.1.1.1 is answering



1 . interception machinery

identify process loading libssl.so

retrieve TLS master secrets

2 . transparent proxy

connect to the real destination

decrypt TLS traffic



- 1 . detect when libssl.so is opened**
check the filename and write PID to a map
- 2 . redirect to a custom proxy**
get PID candidates from the map
rewrite source port with the PID
store real destination in a map
- 3 . get the TLS master secret**
send it to a map indexed by PID



1 . connect to the real destination

retrieve the destination IP using the source port

2 . retrieve the master secret

decrypt traffic!

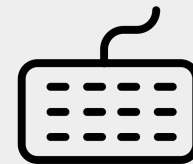
3 . dissect TLS messages

using Scapy

Peetch Demo #5

Transparent OpenSSL Traffic Interception & Decryption

Work In Progress



Created by Sarah
from Noun Project

```
01 # peetch proxy
02 [!] Proxying OpenSSL traffic
03 [+] Decrypting traffic to 208.97.177.124/443 via 127.0.0.1/33305
04 <-- 10.211.55.10 > 208.97.177.124 tcp
05 --> 208.97.177.124 > 127.0.0.1 tcp
06 --> 208.97.177.124 > 127.0.0.1 tcp
07 --> 208.97.177.124 > 127.0.0.1 tcp
08 <-- 10.211.55.10 > 208.97.177.124 tcp
09 --> 208.97.177.124 > 127.0.0.1 tcp
10 <-- 10.211.55.10 > 208.97.177.124 tcp
11 --> 208.97.177.124 > 127.0.0.1 tcp
12 <-- 10.211.55.10 > 208.97.177.124 tcp
13
14 ###[ TLS Application Data ]###
15 data      = 'GET /?name=highly%20secret%20information HTTP/1.1\r\nHost: www.perdu.com\r\nUser-Agent:
curl/7.68.0\r\nAccept: */*\r\n\r\n'
```

Looking Ahead



powerful kernel and userspace hooks

available in most Linux versions

rich ecosystem

bcc, bpftrace...

new security tools ahead

eBPF simplifies putting ideas into code



secdev/scapy

NSS Key Log support

PCAPNg writing & comment option

iovisor/bcc

CGROUP_SOCKET_ADDR program type support

quarkslab/peetch

the eBPF & TLS playground

Questions?

