Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

Analysing repacked Telegram and Signal with use of observability and security tools

Diploma thesis

2024 Ing. Bc. Jakub Škoda

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

Analysing repacked Telegram and Signal with use of observability and security tools

Diploma thesis

Study programme:	Computer Science
Field of study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Daniel Olejár, PhD.
Consultant:	Mgr. Peter Košinár

Bratislava 2024 Ing. Bc. Jakub Škoda

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Ing. Jakub Škoda
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Analysing repacked Telegram and Signal with use of observability and security tools

Annotation: The aim of the diploma thesis is to analyze repacked version of Signal and Telegram instant messaging applications using various observability and security tools. Main purpose of this thesis is first to efficiently recognize if an application uses either Telegram or Signal in the background, to verify if it performs the same function that legitimate original application does and to highlight any non-standard and possibly malicious behaviour.

Aim: The aim of the diploma thesis is to analyze repacked version of Signal and Telegram instant messaging applications using various observability and security tools.

Keywords: repackaging attacks, malware, observability tools, security tools, black-box analysis, behavioral analysis, Signal, Telegram, Linux, Wireshark, strace

Supervisor:	doc. RNDr. Daniel Olejár, PhD.
Consultant:	Mgr. Peter Košinár
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	08.01.2024
Approved:	10.01.2024

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

I would like to thank my supervisor doc. RNDr. Daniel Olejár, PhD. for helping me with the choice of topic and all the academic advice.
I also want to thank my consultant Mgr. Peter Košinár for always finding time for me, productive discussions and friendly approach.

Abstract

ŠKODA, Jakub: Analysing repacked Telegram and Signal with use of observability and security tools. [Diploma thesis] – Comenius University Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Computer Science. – Supervisor: doc. RNDr. Daniel Olejár, PhD., Consultant: Mgr. Peter Košinár, Bratislava: FMPH UNIBA, 2024, 54 p.

The aim of the diploma thesis is to analyze repacked version of Signal and Telegram instant messaging applications using various observability and security tools. Main purpose of this thesis is first to efficiently recognize if an application uses either Telegram or Signal in the background, to verify if it performs the same function that legitimate original application does and to highlight any non-standard and possibly malicious behaviour. In Chapter 1 we introduce past and current examples of repacking attacks and define functioning of the attack. We show that it is a frequently used method of distributing malware especially for Android operating system. In Chapter 2 we focus on various observability and security tools in Linux ecosystem, clarify the distinction between observability tools and security tools. We also briefly outline how these concepts used mainly in GNU/Linux environment adapt to Android. In Chapter 3 we review previous analysis of Telegram, Signal and other instant messaging apps, highlight most important findings and what has changed since their publication. We especially focus on methodologies used and which part of them we plan to reuse or change. Finally, in Chapter 4 we compare official signal-desktop application with unofficial signal-cli to find out whether signal-cli is a legitmate application or does some nefarious activity.

Keywords: behavioral analysis, black-box analysis, eBPF, Linux, malware, observability tools, repackaging attacks, security tools, Signal, strace, Telegram, Wireshark

Abstrakt

ŠKODA, Jakub: Analýza prebaleného Telegramu a Signalu pomocou pozorovacích a bezpečnostných nástrojov. [Diplomová práca] – Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra informatiky. – Vedúci diplomovej práce: doc. RNDr. Daniel Olejár, PhD., Konzultant: Mgr. Peter Košinár, Bratislava: FMPH UNIBA, 2024, 54 s.

Cieľom diplomovej práce je analyzovať prebalené verzie aplikácií na okamžité zasielanie správ Signal a Telegram pomocou rôznych nástrojov na sledovanie a zabezpečenie. Hlavným cieľom tejto práce je najprv efektívne rozpoznať, či aplikácia na pozadí používa Telegram alebo Signal, overiť, či vykonáva rovnaké funkcie ako legitímna pôvodná aplikácia a upozorniť na prípadné neštandardné a prípadne škodlivé správanie. V kapitole 1 predstavujeme minulý a súčasný útok na prebalenie a definujeme fungovanie útoku. Ukazujeme, že ide o často používanú metódu distribúcie škodlivého softvéru najmä pre operačný systém Android. V kapitole 2 sa zameriame na rôzne nástroje na pozorovanie a zabezpečenie v ekosystéme Linux, objasňujeme rozdiel medzi nástrojmi na pozorovanie a bezpečnostnými nástrojmi. Stručne tiež predstavíme, ako sa tieto koncepty používané najmä v prostredí GNU/Linux prispôsobujú systému Android. V kapitole 3 uvádzame predchádzajúcu analýzu aplikácií Telegram, Signal a ďalších aplikácií na okamžité zasielanie správ. Zdôrazníme najdôležitejšie zistenia a to, čo sa odvtedy zmenilo. Zameriavame sa najmä na použité metodiky a na to, ktorú ich časť plánujeme opätovne použiť alebo zmeniť. Na záver, v kapitole 4 porovnáme oficiálnu aplikáciu signal-desktop s neoficiálnou signal-cli, aby sme zistili, či je signal-cli legitímna aplikácia alebo vykonáva nejakú nekalú činnosť.

Kľúčové slová: analýza čiernej skrinky, analýza správania, bezpečnostné nástroje, eBPF, Linux, malware, nástroje pozorovateľnosti, prebaľovací útok, Signal, strace, Telegram, Wireshark

License

Contents

Introduction **9**

1 Existing repackaging attacks **11**
1.1 Rise of repackaging attacks on Android 11
1.2 Recent repacking attacks 14

2 Observability and Security Tools **17**
2.1 Observability tools 19
2.1.1 strace 21
2.1.2 tcpdump and Wireshark 25
2.1.3 eBPF 26
2.2 Pitfalls of Observability tools 30

3 Previous analyses of Signal and Telegram **33**

4 Signal analysis **36**
4.1 BCC tool 37
4.2 connect 37
4.3 open 39
4.4 exec 41

Conclusion **46**

References **48**

List of Figures

1 Apps sampled in [3] by market, ©Gibler et al. 2013 12
2 Apps sampled in [3] by category, ©Gibler et al. 2013 13
3 Security relevant traceable events [12], ©Brendan Gregg/Netflix 2017 . . . 18
4 Brendan Gregg, 2021, CC BY-SA 4.0 International 20
5 Tracing enviroment 21

List of Tables

1 Network connections for signal-desktop 38
2 Network Connections for signal-cli - send message 39
3 Network Connections for signal-cli - Receive all messages 39
4 Network Connections for signal-cli - Receive message from single contact . 39
5 Network Connections for signal-cli - Link 40
6 exec syscalls of signal-desktop 42
7 exec() syscalls for signal-cli send 44

Introduction

This thesis is concerned with detection of repackaged legitimate instant messaging applications by analysing how they interact with their environment (system calls, internet packets), while treating the apps themselves as a black box.

There are two ways to investigate and detect malicious behaviour of software applications. Either you can get the original source code for the software, usually by reverse engineering, analyse the code and see if it includes any malware-like pattern for example file operations such mass encrypting, deletion, sending or altering files that the given application has no reason to access, running code or downloading payloads that seem to have nothing to do with the claimed purpose of the software.

The other option that we will look into in this thesis is observing a behaviour of the software, using various observation tools further explained in Chapter 2, and see if the software does something it is not supposed to, or at least is suspicious.

We will use this observation approach where we treat software as a black box, inside which we cannot see, to analyse and detect instances of repacking attack. That is practice of taking legitimate application and inserting malicious code inside it and then offering it to the public. Sometimes the attacker poses as the original vendor of the application, other times it is offered as an improved version of the original software. We explore recent instances of these attacks in Chapter 1.

Specifically, we will try to analyse various versions of Signal and Telegram. Both Telegram and Signal are popular instant messaging applications. Signal is well regarded for its security and open-source approach - everything from text through calls to gifs that you send are end-to-end encrypted by default, even amount metadata that is visible to Signal servers is limited to bare minimum. Telegram is mostly open-source as well, offers optional end-to-end encrypted chat and most importantly is a frequent target of the recent repacking attack.

First, we build standard behaviour pattern of legitimate versions of both Telegram and Signal. We analyse and document system calls they use, internet packets that they send

and receive as well as any other distinct interaction that observation tools we use will be able to notice. Then, we do the same observation and analysis for modified applications and compare the results. In the end, we try to build an automatized way to detect these modified apps.

Readers should be already familiar with Linux operating system, basics of malware terminology and networking.

1 Existing repackaging attacks

Repackaging attack is use of repackaged versions of legitimate applications with malicious payloads. First part of the attack consists of obtaining a legitimate app, disassembling it enclosing additional (often malicious) payloads, and then re-assembling. For open source and source-available software, or in cases when source code is leaked, situation is even easier as the whole reverse engineering part can be skipped. In the second part of the attack targeted users must be tricked or enticed to download and install modified (malicious) app instead of the legitimate version. [1], [2]

Not all of these repacked applications are malicious. Many are form of application plagiarism, where paid or simply popular apps are repacked, paid advertisements and in-app purchases are included. Such plagiarized applications are then distributed by the repackager to generate income. [2], [3] Authors that are targeted lose on average 14% of their advertising revenue to application plagiarism assuming the users who downloaded the plagiarized versions would have used the original app instead. However, sometimes repacking might be even completely legitimate such as rebranded apps (apps from the same developer with high code similarity) [3] or just white label apps or a fork of existing open source projects.

1.1 Rise of repackaging attacks on Android

Repackaging attack was popular during the rise of Android mobile operating system. In 2012 [1] callected 1,260 malware samples from various Android markets - stores which distribute and sell applications for Android operating system. Out of these 1,083 (or 86.0%) were repackaged versions of legitimate applications with malicious payloads.

In 2013 [3] crawled 265,359 free applications from 17 Android markets around the world (73.7% of the apps are from the official Google Play market, 14.7% are from 9 third-party English markets, 13.8% are from 6 third-party Chinese markets, and 0.46% are from 2 Russian markets). Out of the sampled apps, 44,268 (16.7 %) were cloned apps. Authors define cloned application as “an app that is a modified copy of another app, and thus shares a significant portion of its application code with the original”, which is rather

interchangeable with our term “repacked apps”.

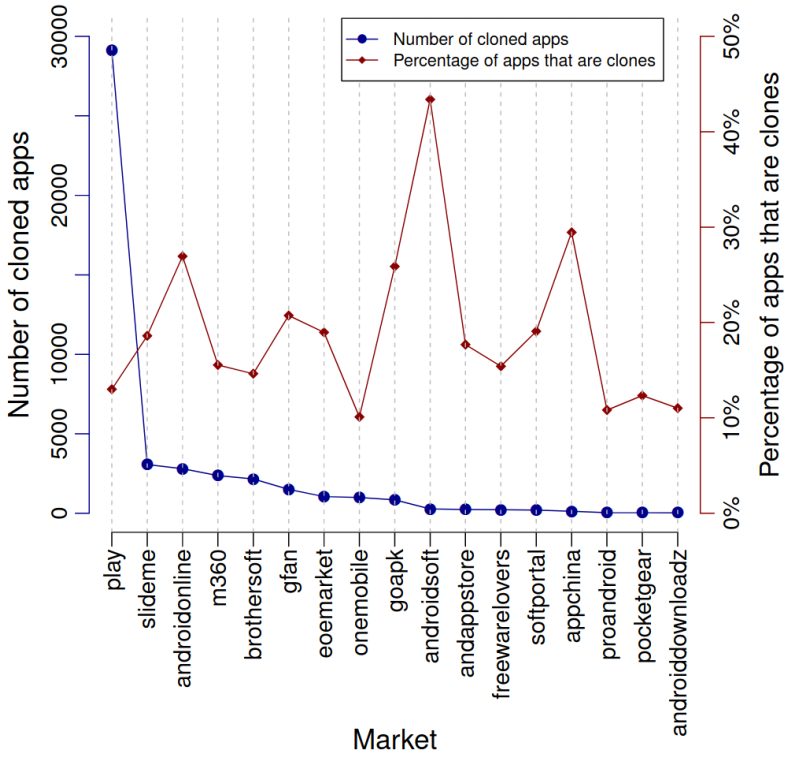


Figure 1: Apps sampled in [3] by market, ©Gibler et al. 2013

Figure (1) showing the popularity of different app markets in clone clusters by [3]. The absolute number of clone apps from each market is represented by the axis labelled “Number of cloned apps”. The axis labelled “Percentage of apps that are clones” is simply result of number of cloned apps from a specific market divided by the total number of apps from that market in the entire database.

As most of the sampled apps came from Google Play, around 195,570 out of 265,359, it is not surprising that in absolute terms the bigger number of cloned apps was identified here. Almost 30,000 cloned apps, which is somewhat over 10% of sampled Google Play apps. However, in relative terms Google Play had one of the lowest percentage of cloned apps. Outside Google Play, [3] sampled around 69,790 apps. Out of these samples AndroidSoft had most cloned apps, almost 50% of the sample. It was followed by Chinese markets AndroidOnline, GoApk, and AppChina, that had each almost 30% of cloned apps in their respective samples.

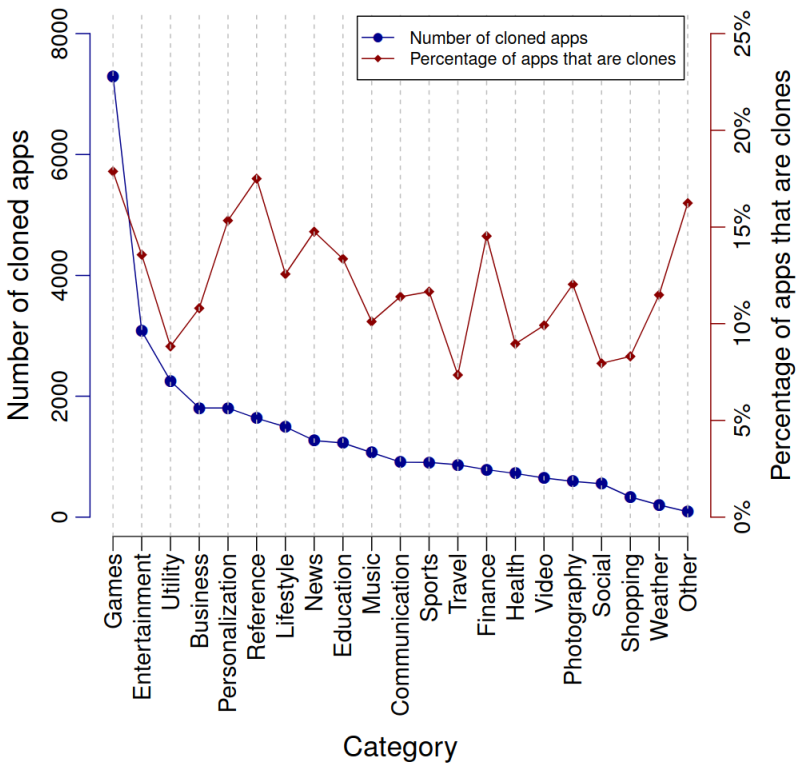


Figure 2: Apps sampled in [3] by category, ©Gibler et al. 2013

Figure (2) is fairly similar to the previous one, just displays apps based on category, what they are used for, instead of market. "Number of cloned apps" axis represents the absolute number of cloned apps in each category. "Percentage of apps that are clones" is result of number of cloned apps for a specific category divided by the total number of apps from that category in the entire database.

Games are the most frequently cloned category with in both absolute terms, over 7,000 apps, and relative terms, around 18% of games in the database were clones. Other popular category is Reference, which includes Books & Reference, E-books, Ebooks & Reference, Reference, with around 17% of cloned apps. And also, a vague category of Other with around 16%, which include Developer/Programmer, Home & Hobby, Other, Religion. Then Personalization, which includes Personalization and Wallpapers, with above 15% of cloned apps. Rest of the categories had between 7% to 15% of clones, namely the category Communication had around 11% of clones.

1.2 Recent repacking attacks

In the last five years trend of malicious repacked app continues on Android. As security measures on Google Play tighten, attackers can now be frequently seen persuading users to download malicious apps from sources outside of applications markets such as website mimicking legitimate website of the app or Telegram channels. However, there are still numerous examples of malicious repacked apps on Google Play and other stores. Frequently their aim is to steal cryptocurrency funds or just a part of more focused spear-head attack.

In 2023 there was a large group repackaged Telegram and WhatsApp apps targeting mainly Chinese users as both apps are banned in China. Threat actors set up Google Ads leading to fraudulent YouTube channels (reported by authors of [4] and now taken down by Google), which included links to copycat Telegram and WhatsApp websites. [4]

Maliciously repacked versions of both Telegram and WhatsApp targeted cryptocurrency funds using malware called clippers. According to authors this is the first instance of clippers built into instant messaging apps. In addition, some of the maliciously modified Telegram apps used optical character recognition (OCR) to read text in screenshots and photos to steal a seed phrase used for recovering cryptocurrency wallets – this is the first instance of Android malware using OCR. Other monitored Telegram for keywords related to cryptocurrencies and sent the attacker the full message if the keyword is recognized. In some cases, they even exfiltrated internal Telegram data and basic device information. Beside Android attackers also targeted Windows where malicious Telegram used remote access trojans (RATs) that enabled full control of the victim's system. [4]

Clippers mentioned above are a type of malware that steals or modifies the contents of the clipboard. It is well suited for stealing cryptocurrency because addresses of online cryptocurrency wallets are composed of long strings of characters, which are mostly copied and pasted using the clipboard instead of being typed manually. Malware simply switches the victim's cryptocurrency wallet address for the attacker's address in chat communication, with the addresses either being hardcoded or dynamically retrieved from the attacker's server. [4] Current clippers work despite that for Android versions 10 and higher clipboard data can be accessed only by the default input method editor (IME) or the app that

currently has focus. [5]

Many malicious applications are often promoted via various Telegram Channels. This is also the case of WhatsApp mod that is offering users additional features, which was not malicious from the start, but later became infected trojan with called Trojan-Spy.AndroidOS.CanesSpy. It was active from mid-August 2023 to at least around October 2023. It was promoted on Telegram channels, where the most popular channel had almost two million subscribers. Telegram channels were mostly in Arabic and Azeri languages and impacted regions were mainly Azerbaijan, Saudi Arabia, Yemen, Turkey, and Egypt. [6]

In some cases, a repacked application can pose as a completely different service. This is the case of trojanized Telegram active in November 2021 that included StrongPity backdoor code and posed as an official app of random-video-chat service Shagle. Shagle is a real service that provides encrypted communications between strangers, however, Shagle's service is only accessible through their official website and Shagle does not provide an Android app. Malicious app was distributed via copycat website, mimicking the Shagle service. The StrongPity backdoor is modular, all necessary binary modules are encrypted using AES and downloaded from C&C server. It has 11 dynamically triggered modules, which are responsible for recording phone calls, collecting SMS messages, lists of call logs, contact lists, and much more. [7] StrongPity is name of used malware as well as name of the group responsible for the attack. More frequently they use the name PROMETHIUM, and are focused on espionage. They have been active since at least 2012. [8]

Google Play is still viable channel for repackaged malware as well, especially for various Telegram clippers. Other applications also appear such as malicious repacking of WhatsApp and Signal as well as of crypto wallets and VPNs.

From June 2020 to January 2021 there was malicious version of Telegram under name FlyGram on Google Play and got over 5,000 installations. It was also available on Samsung Galaxy Store and dedicated websites. Using Android BadBazaar espionage code can extract basic device information, contact lists, call logs and the list of Google Accounts. Surprisingly, it can extract only limited information and settings related to Telegram. Telegram's contact list and messages are not included unless users enable a specific FlyGram feature that

allows them to back up and restore Telegram data. This sends back up data to a remote server controlled by the attackers, giving them full access excluding only the collected metadata. [9]

From July 2022 to May 2023 there was malicious version of Signal called Signal Plus Messenger on Google Play. It used the same Android BadBazaar espionage code and distribution channel as Flygram. Signal Plus Messenger seems to have been more focused on spying on communication than Flygram. It can extract the Signal PIN number that protects the Signal account, and it misuses the link device feature that allows users to link Signal Desktop and Signal iPad to their phones. [9]

Other Telegram based malware on Google play had descriptions in traditional Chinese, simplified Chinese and Uighur, telling us a lot about targeted region. They included full-fledged spyware capable of stealing the victim's entire correspondence, personal data, and contacts. Code of these malicious apps was only marginally different from the original Telegram code, which helped them pass the security checks to get to Google Play. [10]

2 Observability and Security Tools

A slight complication with repacking attack is that knowing the application is repacked is not enough. The application in suspicion often openly claims to be a modified version of an original and sometimes such modifications are made with a genuinely good intent, non-malicious and even useful. Application must be caught red-handed to proof that it truly is a case of repacking attack.

There are two ways to investigate and detect malicious behaviour. Get the code for both original and modified app and compare them, see if there is any malware-like code in the modified part. Or observe behaviour of the app and watch if it does something that original app never does.

The first approach usually involves reverse engineering as often source code is not available. Even if the source code for modified app is available, it cannot be trusted (unless Reproducible builds/deterministic compilation is used) as attacker could be including additional malicious code into a final build that they are not sharing. The biggest drawback is that you have to be well-versed in language that the application is written in, which might be a problem if you plan to investigate wide range of different applications. Sometimes after updates parts of application are rewritten in different, for example Signal moving protocol, cryptography and rest of the backend libraries from Java to Rust. [11]

With the second approach we try to identify anomalous or unexpected behaviour of the repackaged applications. This includes monitoring system calls, network traffic, and file interactions which are crucial for understanding the runtime activities of repackaged applications. Unlike reverse engineering, which often involves static analysis of compiled code, observability tools allow dynamically monitor the execution of repackaged applications. This allows us to observe the application's behaviour in various contexts, including interactions with the operating system, network, and other applications.

However, it is most likely not feasible to observe and then to analyse all aspects of application's behaviour. We need to choose aspects of the behaviour that we wish to watch. Figure (3) provides a diagram of place in operating where we can look and examples of

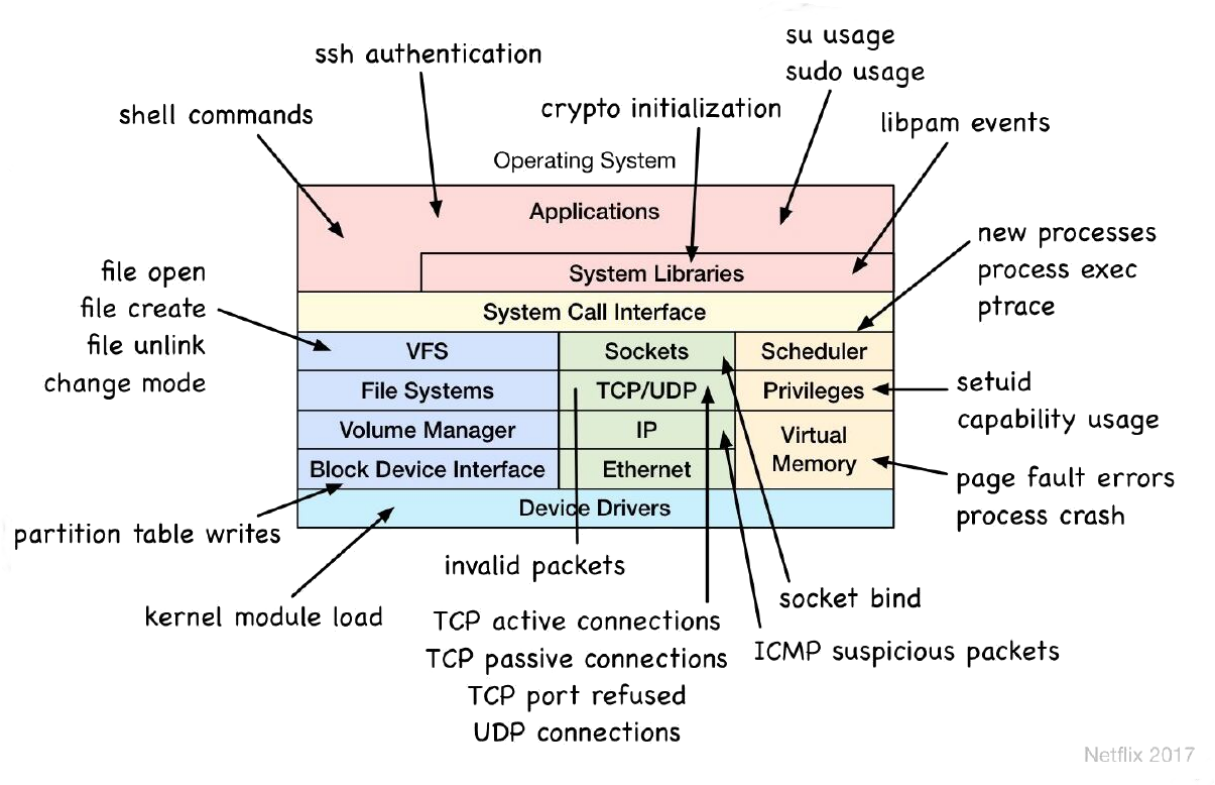


Figure 3: Security relevant traceable events [12], ©Brendan Gregg/Netflix 2017

what kind of events we find there. Interesting events depend on the application, however, checking interaction with files, internet connection and used process forms a good basis.

Traffic analysis is represented in the diagram by the green stack. It lets us identify communication patterns between repackaged applications and external servers or command and control servers. Analysing network traffic can identify connections to malicious servers, data exfiltration attempts, or other network-based attack vectors. To limit overheads it is important to track a low-frequency event, especially in context of security. Therefore, it is better to track TCP connection init rather than TCP send/receive.

VFS (Virtual Filesystem Switch) handles system call related to file manipulations. By observing it, it can tell us about files that are opened, copied or deleted. Maybe it is reading files that it is not supposed to or maliciously modifying configuration files.

System call interface tells us when processes are launched. Here we can also observe when application requests something from a kernel via a system call.

Watching the application layer itself we can see shell commands, `ssh` authentication, crypto initialization, `sudo` usage, `su` usage. Usage of system libraries is easy to trace as the `libpam` events has a good API for that. Interesting might be the usage of cryptographic libraries as they might be use by ransomware or in decryption of obfuscated malicious code, however, we need to keep in mind that most internet connected applications use cryptography nowadays for secure communication.

Below we review a few of the selected observability tools and explore some part of their inner workings; then pitfalls of observability tools and why specialized security tools are needed. Finally, we introduce selected security tools.

2.1 Observability tools

In computer engineering the word observability is used to describe the tools (for reading state), data sources (metrics and logs), and methods for understanding (observing) how a technology is operating. Unlike benchmarks and other performance tools that change the state of the system to understand it, observability tools look at the system ideally without changing it. [13]

A perfect observability tool would just look at the system without touching at all. However, real tools still have impact on the system. Their execution consumes resources, usually negligible, but in some cases it is enough to perturb the target of study. [13]

We can split Linux tracing systems into **tracing frontends** (the tool we actually interact with to collect/analyse data), **mechanisms for collecting data** for the frontends and **data sources** (where the tracing data comes from).

Tracing frontends are command line or GUI applications in which user types commands and gets corresponding output. Wide variety of them is listed in Figure (4) as the bold text next to start of the arrows. As we can see from the diagram, each tool has its specialized use and focuses on tracing different part of operating system. There are also some more general tools mentioned on the left of the diagram (perf, Ftrace, BCC, bpftrac, LTTng) that can observe many parts of the operating system.

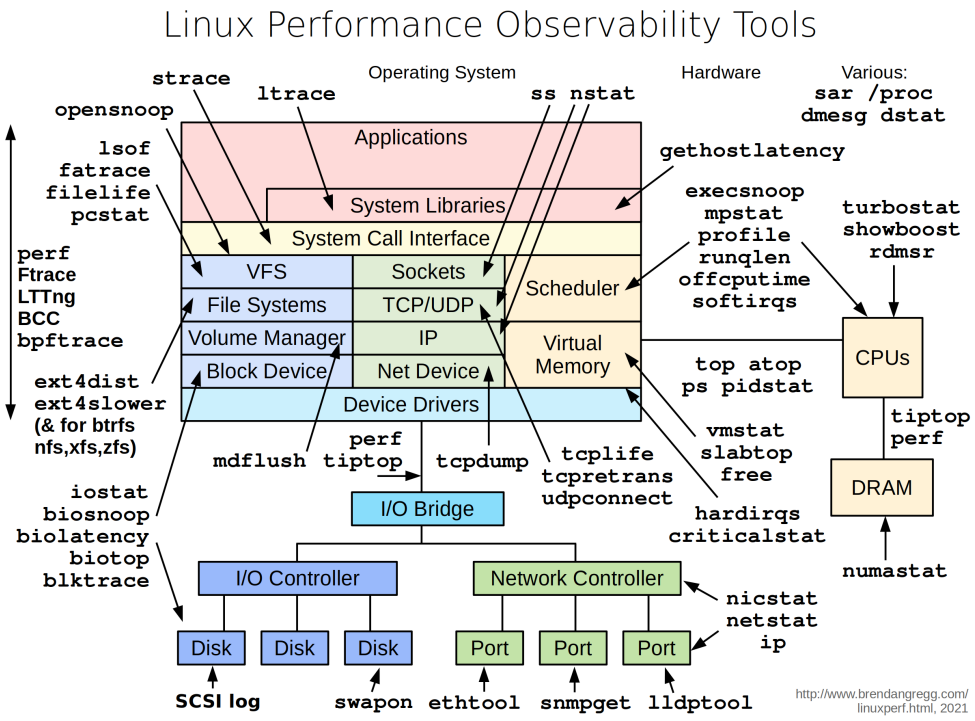


Figure 4: Brendan Gregg, 2021, CC BY-SA 4.0 International

Mechanisms for collecting data get the data from the data source and serve them to the front end. Only `ftrace`, `perf` and `eBPF` are official part of Linux kernel being integrated directly into the Linux kernel source code. While `LTTng`, `dtrace`, `SystemTap` or `sysdig` work as out of tree Loadable Kernel Modules, which are developed and maintained separately from Linux kernel and kernel can load and link these Loadable Kernel Modules at runtime. Beside `eBPF` all here mentioned share names with their respective tracing frontend, eBPF frontends includes BCC and bpftrace. Many of these mechanisms also use other mechanisms for collecting additional data as you can see in Figure (5). For example `perf` uses its own sysem call `perf_open()` as well as tracefs, a specialized file system used by `ftrace`. Many of these forntends are also currently working on implementing use of BPF.

There are just four official data sources: kprobes, uprobes, tracepoints, perf_events. However, the way that the mechanism processes these data sources can differ drastically, for example eBPF does all data processing inside the kernel and only sends final filtered result to the user-space, which makes it highly efficient.

First, we look at two commonly used tools. For looking at System Call Interface we will use

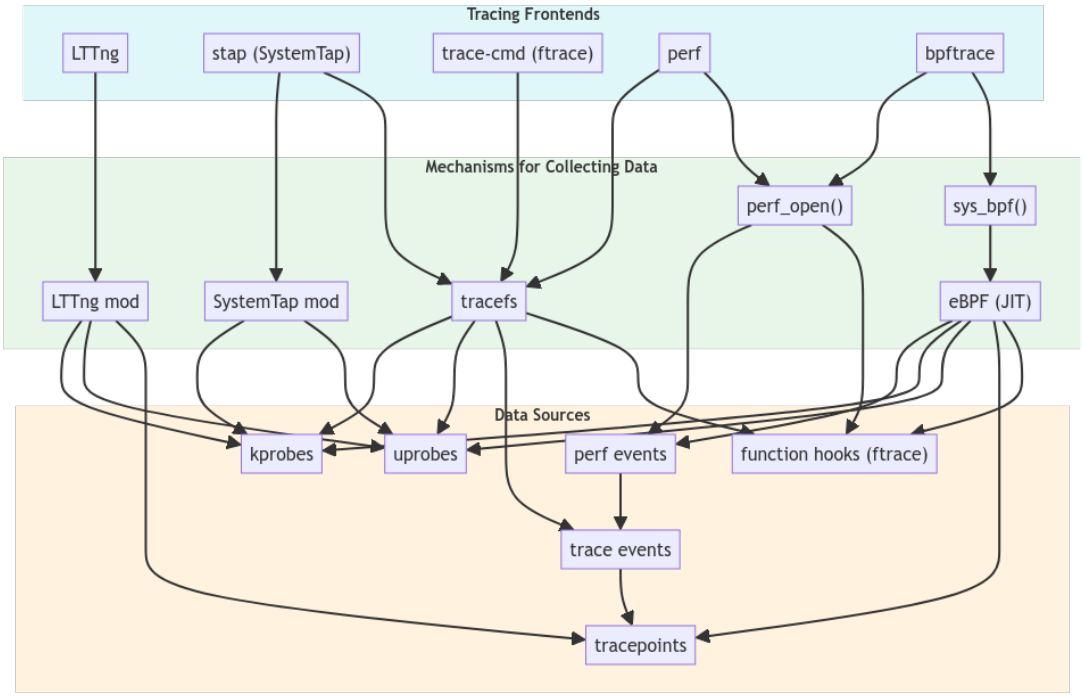


Figure 5: Tracing enviroment

`strace`. For Net Device we will use `tcpdump` which is internally also used by Wireshark. After exploring some parts of their inner workings and related drawbacks we will look at some more efficient tools that are using eBPF or perf.

For more detailed overview of observability tools see [14] and [15] For history overview see [16] and for minimal code examples see [17].

2.1.1 strace

`strace` is a system call tracer for Linux. It shows us most of the actions that are in Figure (4) under "System Call Interface". It was developed by Paul Kranenburg in 1991 for SunOS [18] and ported to Linux a year later.

Kernel feature known as `ptrace()` (process trace) is used by `strace` to pause the target process at the beginning and the end of each syscall, so `strace` can read state. This way that `strace` works in the background, is also its biggest drawback. Pausing an application twice for each syscall, and context-switching each time between the application and `strace` slows down the whole application considerably. [19] There is an infamous example of strace

making process 42 times slower.

```
$ dd if=/dev/zero of=/dev/null bs=1 count=1M 2>&1 | grep -v records
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.668768 s, 1.6 MB/s


$ strace -f -qq -e signal=none -e trace=fchdir \
dd if=/dev/zero of=/dev/null bs=1 count=1M 2>&1 | grep -v records
1048576 bytes (1.0 MB, 1.0 MiB) copied, 28.0445 s, 37.4 kB/s
```

Thankfully, strace is still in active maintenance and evolving. Many improvements involve limiting `ptrace()` syscalls or replacing them completely. An older but clear example of this is 2012, when they started to use `process_vm_readv` instead of `PTRACE_PEEKDATA` to read data blocks such as filenames and data passed by I/O syscalls. `PTRACE_PEEKDATA` gets one word per syscall, which is very expensive. For example, in order to print fstat syscall, we need to perform more than twenty trips into kernel to fetch one struct stat. Now used `process_vm_readv()` can copy data blocks out of process address space. For more examples of such small improvements see [20].

Major speedup occurred in version v5.3 (September 2019), when they added Seccomp-assisted (Secure Computing Mode-assisted) system call filtering, which automatically generates and attaches a BPF program to filter system calls. This makes execution of untraced system calls (those that we did not specify in `strace` filter) two orders of magnitude faster. [21]

We get this speedup as system calls are allowed to run more or less normally with use of ptrace_cont (ptrace continue), only once we have syscall of interest seccomp-stop stop the programme, strace is called and then the process is restart with ptrace_syscall. The `dd` process that we traced before is with use of `--seccomp-bpf` flag only 1.10 times slower than the same untraced process instead of 42 times slower as it was before. [21]

```
$ strace --seccomp-bpf -f -qq -e signal=none -e trace=fchdir \
dd if=/dev/zero of=/dev/null bs=1 count=1M 2>&1 | grep -v records
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.736511 s, 1.4 MB/s
```

However, to make further improvements in `strace`, `ptrace()` itself must be improved. This is problematic as already in 2010 development of `ptrace()` was considered "frozen". There were new features or improvements added, only bugfixes. https://lwn.net/Articles/371505/ Therefore, any improvement of `ptrce()` seems unlikely. There were also attempts to replace `ptrace()` completely with something new. Doing so seems even more difficult as `ptrace()` is a standard function of Linux kernel Applications Binary Interface. This cannot be removed from the kernel, unless "everybody" stops using it. This would mean that at first "everybody" would have to replace `ptrace()` with working alternative – be it in standard tool such as `strace` or GDB, as well as custom proprietary code used internally in some major companies, and only after that it could be removed from kernel. This would require to include alternative to `ptrace()` in kernel, while keeping `ptrace()` there as well. Doing that is possible but it is a really unpopular option among Linux kernel developers. It would results in need to maintain two tracing interfaces instead of just one. [22]

This whole situation resulted in deadlock in `ptrace()` development, which goes almost completely standstill. This would not be a problem if `ptrace()` was good enough, however, `ptrace()` is far from that. The `ptrace()` system call is considered to be one of the worse parts of the Unix interface due to inelegant and complicated design. The design problems include strange semantics (it reparents the traced process to the tracer), so badly defined interactions with jctl (job control) that the current behaviors are broken to the point where achieving transparency with userland work-arounds is impossible. For example, a task which is running under `strace` can be stopped with `^Z` as usual, but the shell will be unable to restart it as tracers have currently no way to know that the real parent has tried to start a stopped process. [24].

The general purpose of ptrace() is to allow one process to monitor and modify the state of another. It was originally intended for interactive debuggers such as GDB debugger but now is used in various sandboxing schemes, in `strace` as well as for internal management of user-mode Linux. [22]

Whenever system calls have to work with extended state within the kernel the preferred mechanism, which `ptrace()` does **not** use, for referring to that state in user space is

the file descriptor. With file descriptors there are well-defined mechanisms for event multiplexing and system calls behave "naturally". Instead of using file descriptors, `ptrace()` unfortunately depends on rather arcane mechanism. A process to be traced is removed from its normal place in the process tree; the process doing the tracing becomes its new parent. In other words, `ptrace()` sets up a sort of temporary foster home for children under scrutiny. The new parent can then learn about events in the child through the `wait()` system call. [22]

In Linux process can only have one parents [25], because of this any given process can be traced by only one other process at any given time. Usually this is not a problem as rarely someone debugs a process with two debuggers at once. However, this limitation often causes problems for developers of debugging tools. Because `strace()` is also used for sandboxing using `strace` or debugger is often not possible either. [22]

This limitation also allows to check if they are debugged or observed with `strace` or debugger. Malicious bug can just call `ptrace()` on its own malicious process and if `ptrace()` fails simply assume that is already observed by someone else `ptrace()` and refrain from malicious action during the time of observation. [26] As `ptrace()` allows control over processes, it also has malicious uses and was used in several real-world attacks and exploitations. This includes the DirtyCow [27] bug exploitation. CVE-2011-4327, where `ptrace()` allowed local users to obtain sensitive key information. And also remote access trojan called Puppy uses `ptrace()`. [28]

Furthermore, `ptrace()` system call is also defined as a complex, multiplexer call (see the man page for details) which is hard to understand and hard to use efficiently. User-space code which uses `ptrace()` tends to become encrusted with non-portable workarounds as `ptrace()` is hard to implement correctly and consistently. [22]

For more information about `strace` we recommend illustrated "strace zine" [29] and critical introduction with list of many drawbacks by [31].

`strace` is a great tool held by arcane `ptrace()`. For behavioural analysis it might be problematic that `strace` is fairly easy to detect and cannot be used on many sandboxed

applications. There are multiple alternatives to `strace` use eBPF such as `opensnoop`, or `perf` system calls and other some of which we mention below. In the future `strace` itself might start to use eBPF and solve most of it `ptrace` related problems with that.

2.1.2 tcpdump and Wireshark

Tcpdump is a tool for capturing network traffic and packet analysis. It was developed in 1992 by Steven McCanne and Van Jacobson for SunOS as alternative for default `etherfind`. `etherfind` was based on Unix find command and it was getting the computer overwhelmed because its user-space process was unable to process all incoming packets. `tcpdump` solved this problem by filtering packets with in-kernel virtual machine BPF (see Chapter 2.1.3) that was developed alongside `tcpdump`. At the end of the development the compiler system and filtering engine was put out of the `tcpdump` and an API and reusable library called libpcap was created. A common file format called `pcap` was also created, which as an elaboration of `tcpdump` flag `-w`, write data into a file, such as `tcpdump -w http.pcap port 80`. [32]

Beside `tcpdump`, the most prominent project using libpcap is Wireshark. All Wireshark tools use libpcap for packet sniffing and can also use `tcpdump` filter syntax as capture filter. [33] Wireshark in addition offers more advance filtering, features as connecting TCP packets from the same connection and GUI. As Wireshark uses same library as `tcpdump`, it is possible to capture packets with `tcpdump -c 100 -w my_packets.pcap dest port 8080` and than later open it with Wireshark using `wireshark -k -i -`. This might be especially useful if we are capturing packets on remote host where Wireshark is not installed, where we can use something like `ssh some.remote.host tcpdump -pni any -w - -s0 -U port 8888 | wireshark -k -i -`.

If for some reason you do not want to or cannot use `tcmpdump` but still have just a command line application with no GUI, Wireshark come with collection of terminal tools as well. There is TUI version of Wireshark called `tshark`. Both `tshark` and Wireshark have mostly equivalent functionality when, just Wireshark's GUI user can select which packets to save, `tshark` will record everything.

Then `tshark` itself runs another command line tool from Wirshark developers called `dumpcap`, which is a small program whose only purpose is to capture network traffic, while retaining advanced features like capturing to multiple files. In most cases you will just `tshark` instead, [34] but as `tshark` just runs `dumpcap`, their performance should be the same. [35]

For more information about Wireshark and tcpdump see [36].

2.1.3 eBPF

BPF was developed by Steven McCanne and Van Jacobson alongside `tcpdump` as its kernel module, a virtual machine model that would run in the kernel. Now eBPF makes the Linux kernel programmable without need for modifying kernel code or creating kernel modules. Thanks to this eBPF tools form a basis of many versatile and efficient observability tools such as bcc, bpftrace or Cilium. [37]

In 1992 BPF was introduced as BSD Packet Filter, a pseudomachine that can run filters to determine whether to accept or reject a network packet. Filters for BPF are written as programs using the BPF instruction set, a general-purpose set of 32-bit instructions that closely resembles assembly language. [38], [39]

Bytecode as written in 1992 article The BSD Packet Filter: A New Architecture for User-level Packet Captur

```
ldh     [12]
jeq     #ETHERTYPE IP, L1, L2
L1:     ret     #TRUE
L2:     ret     #0
```

code output of `sudo tcpdump -d -i wlan0 -n ip`, `-d` flag outputs the bytecode.

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

Both of these examples of instruction set filters out packets that are not Internet Protocol (IP) packets, leaving only IP traffic in the output. The first instruction `ldh` loads a 2-byte value starting at byte 12 in from an Ethernet packet. The instruction `jeq` compares this 2-byte value with the value that represents an IP packet. If it matches, execution jumps to the instruction labeled `L1`, and the packet is accepted by returning a nonzero value `#TRUE`. If it does not match, the packet is not an IP packet and is rejected by returning `0`. [40]

As BPF's name implies, BPF implementation came from BSD with BSD license and it was used as a packet filter. Packet filters make network monitoring more efficient by discarding unwanted packets in kernel and copying to the user-space only data that user is actually interested in. BPF has register-based filter evaluator with a non-shared buffer model. It was meant as replacement for Unix's default stack-based filter evaluator.

Later BPF started to be called Berkeley Packet Filter (BPF). Nowadays, in order to avoid confusion with newer version of BPF, it is also called classic BPF (cBPF). [41]

cBPF got it in Linux in 1997 in version 2.1.75. For long time it was only used as socket filter by packet capture tool tcpdump (via libpcap). [42]

Using the same principle of doing most operations in the kernel and only sending the final result to user space could speed up many other processes beside packet filtering. However, for almost 15 years in-kernel virtual machine went unnoticed. cBPF capability to run untrusted programs in a privileged context was underutilized, despite providing a secure alternative to Kernel modules.

Things started to change in 2011 when just-in-time (JIT) compiler was added for cBPF on x86. This made customized network-packet filtering extremely fast. [16], [43]

First non-networking use case (probably even first outside of libpcap) appeared in 2012, in Linux kernel version 3.5, when `seccomp-bpf` was introduced. It is a system call filtering program that uses a configurable policy implemented through BPF instructions. For example, it is used in Linux version of Chrome as the main layer-2 sandbox, designed to shelter the kernel from malicious code executing in userland. [42], [44] `seccomp-bpf` is improvement upon `seccomp` (SECure COMPuting Mode) introduced into the Linux

kernel in version 2.6.12 (8th March 2005). Whereas `seccomp` restricts the system calls available to a process to only `read`, `write`, `_exit` and `sigreturn`, `seccomp-bpf` uses BPF programs to filter on arbitrary syscalls and their arguments (constants only, no pointer dereference). [45] Additionally, according to The Linux Kernel documentation "BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks". [46] After that BPF was also implemented in Kernel 3.9 as iptables module xt_bpf [47] and in Kernel 3.13 [48] "cls_bpf" classifier for traffic shaping (QoS) [49]. Use of BPF was slowly getting traction.

Developer Alexei Starovoitov in 2014 created extended (eBPF) to make BPF programs applicable to other parts of kernel, including tracing. [16]

In March 2014, eBPF replaced classic BPF (cBPF) in-kernel when it was accepted by David S. Miller, the primary maintainer of the Linux networking stack. Since then the Linux kernel internally translates classic BPF (cBPF) calls into the into the eBPF instructions, in order to provide backwards compatibility. [50], [51]

The eBPF brought major changes compared to classic BPF. The BPF instruction set was completely overhauled to be more efficient on 64-bit machines; the interpreter was entirely rewritten; the eBPF verifier was added to ensure that eBPF programs are safe to run; new data structure for sharing information between BPF programs and user space called 'maps' were introduced; `bpf()` system call was added to the user space; and many other changes. In 2015 ability to attach eBPF programs to kprobes was also added. [40]

This extended use of eBPF beyond packet filtering also caused a formal name change, where eBPF is no longer considered to be an acronym and is used just a proper name, sometimes even just referred to BPF as well.

Interesting topic that is beyond scope of this thesis is why has Linux kernel developer decided to reinvent whole infrastructure of virtual machines, JITs and scripting languages when well establish solutions already exist outside of kernel word. For example, in past there were active initiatives to get Lua in kernel. [52] The answer, why eBPF succeeded where these other initiatives did not, maybe lies in fact that eBPF is based on component that is

already in Linux kernel since 1997. And it is much easier to persuade kernel developers to allow extending this already well tested and trusted tool rather than introducing a completely new thing to kernel.

To make use of eBPF you need BPF bytecode that tells the virtual machine what to do. Thankfully, usually it is not needed to write it directly but there are multiple front-ends with their own scripting languages.

Best start for beginners is `bpftrace`, which is suitable for one-liners and short scripts.

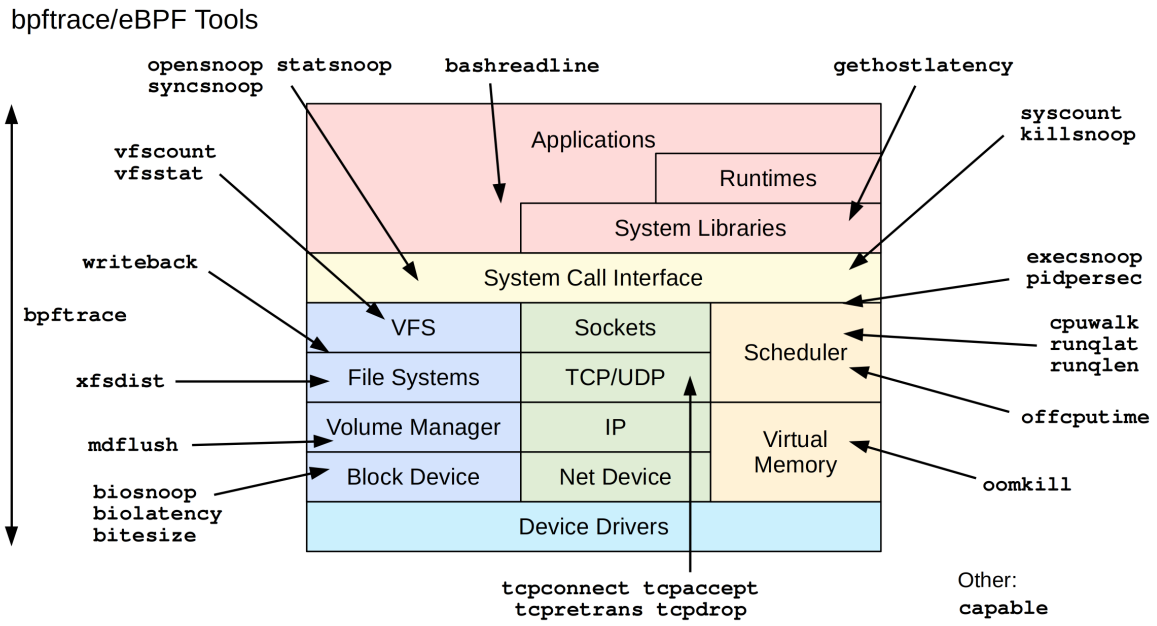
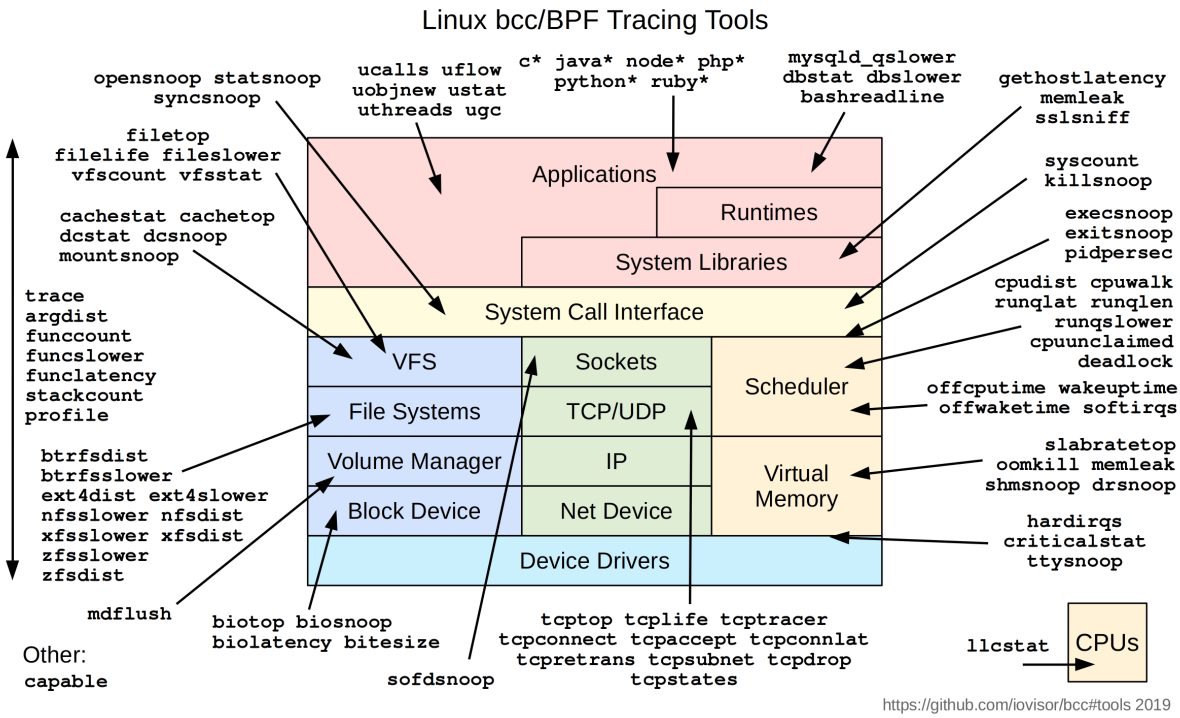


Diagram by Brendan Gregg, early 2019. https://github.com/iovisor/bpftrace

For more complex tools and daemons BCC is recommended. Libraries from `bcc` are also internally used by `bpftrace`.



For many generic tasks there is no need to write own script as many are already readily available, especially for BCC.

To learn more see https://brendangregg.com/ebpf.html and https://andreaskaris.github.io/blog/networking/bpf-and-tcpdump/. For full timeline see https://en.wikipedia.org/wiki/EBPF#History. These two videos can be also helpful to learn more about history of BPF. Videos https://www.youtube.com/watch?v=DAvZH13725I Kernel Recipes 2022 - The untold story of BPF eBPF and Kubernetes: Little Helper Minions for Scaling Microservices - Daniel Borkmann, Cilium https://www.youtube.com/watch?v=99jUcLt3rSk

2.2 Pitfalls of Observability tools

Of course problems with tools above are that they are meant as observability tools not as security tools. While useful for quick initial analysis, they are designed to have the lowest overhead possible, as they are used for analysing live production systems which introduces many trade-offs. Detection by observability tools might be avoided by overwhelming system, time-of-check-time-of-use attacks (TOCTOU) and other techniques.

Even simple `ls(1)` can be avoided by putting escape characters such as `\n` newline character,

carriage return `\r` or backspace `\b` into a filename. `top(1)` relies on "comm" field (the command name) to display process information. An attacker might change the names of malicious processes to resemble innocent or common process names to avoid detection. tcpdump(8) that we use for analysis will drop packets if the system is overloaded, resulting in incomplete visibility. Advanced attacker could overwhelm the system with mostly innocent packets, or wait for the moment when system is overwhelmed, and only sent malicious packets at that time. In a similar way `strace` could be overwhelmed by simple fork bomb `:(){ :|:& };:` or injecting code into legitimate processes or libraries.

As we compare suspicious app to its legitimate version most basic techniques of simply overwhelming system would automatically raise a flag, but it is good to keep this in mind. They secure strict non-repudiation it is possible to set kernel to halt immediately if it could not log an event. There are also less drastic ways such as at least logging the failure of logging as well specialized security tools better suited for such analysis. Adapting observability tools into security tools generally increases overhead by for example adding extra probes. Good security tools need to be built differently. Ideally, security tool should be integrated directly into the Linux Security Modules framework, use a plugin model instead of standalone CLI tools, support configurable policies for event drop behaviour, have optimized event logging and more.

Linux Security Modules hooks are used to add Linux kernel extra security checks and controls that go beyond what the base kernel provides. Being integrated directly into the Linux Security Modules framework leverages the existing security infrastructure in the Linux kernel and integrate security tool more tightly with the system's security mechanisms.

Plugin model involves designing the security tool to be extensible through plugins or modules. Instead of a monolithic standalone tool, a plugin model allows users to add or customize functionality through separate modules. This approach provides flexibility, as users can choose the specific features or modules they need, and it allows for easier maintenance and updates.

Configurable policies for Event Drop Behavior allows users to define rules for when certain events should be dropped or allowed. For example, users might want to configure the tool

to drop specific types of events under certain conditions while allowing them under different circumstances. This can help in mitigating attacks aimed at overwhelming system resources, obfuscating logs, or masking malicious activities. To defend against overwhelming attacks administrators might set thresholds for the number of events per second or per minute. If the incoming events exceed these thresholds, the system could be configured to drop or ignore excessive events, preventing the attacker from overloading the logging system.

Optimizing event logging includes minimizing the impact on system performance, implementing mechanisms to handle high event volumes without losing critical information and ensuring appropriately detailed logs. [53]

Observability tools might at first glance be instantly reusable as security tools. While observability tools might give us a good many pieces of useful pieces of information and sometimes even detect malware, these tools fall easily to purposeful evasion techniques.

3 Previous analyses of Signal and Telegram

Most of the results on what is the legitimate behaviour of the instant messaging application comes from studies that with use of device and network forensic methods tried to learn something about the legitimate applications. They tried to access data and metadata from often encrypted database and determine type of activity based on visible network traffic. Their aim was not to directly asses legitimate behaviour of the application, they assumed that they work with the legitimate version, but to undertake planned forensic analysis they had to identify location in file system, servers that app connects to and other data useful for us.

Most of the available studies focus on the device forensics [54], with the main focus on the internal structure of databases stored on the devices but also including analysis of the file structure and extraction of useful information from artifacts. Analyses of encrypted file structures and databases is less studied [55].

Network forensics traces communications through the network packets to identify the traces of the app's activity. For instant messaging there are already studies that can identify connection establishment, an encryption protocol, and payload sizes for analysis and some can also distinguish behaviour such as calling, texting and similar.

For the more complex literature overview we recommend [56], [57].

Unsurprisingly most available studies are focused on WhatsApp as it is the most used instant messaging application. There are also numerous Telegram studies available [58]. Number of forensic Signal studies is surprisingly low.

Research on Telegram has spanned various aspects, including device forensics across different platforms.

The 2017 study on Telegram for Windows Phone provided a comprehensive overview of the app's architecture and user data file management, which remains relevant beyond the discontinued Windows 10 Mobile platform. It included useful description of general concepts related to Telegram such as the TL (Type Language), names of user data files as

well as detailed guide of how to get information from these data files. [54] Most information in the paper seems to be applicable for other platforms and therefore is useful even though Windows 10 mobile is no longer supported since December 10, 2019 [59] and Microsoft is not developing another mobile operating system. [60]

Another study in 2017 analyzed network traffic from Telegram as well as Facebook and Twitter on the now-obsolete Firefox Mobile OS, revealing insights into the app's server connections and data transmission patterns. Firefox Mobile OS simulator was used. Generated traffic was sniffed by Wireshark, resulting in the list of IP addresses, ports, domains, and subdomains used by these apps. Key takeaways for Telegram are that generating the registration key Telegram connects to `us-west-2.compute.amazonaws.com` (analysis was done in the USA therefore domain name ends with `us-west-2`, this would change depending on the location) which was used to push and generate the registration key for Telegram, and unsurprisingly to `telegram.org`. After registration Telegram mostly connects just to `telegram.org` but also received traffic from `github.map.fastly.net` when received a message from another contact. When sharing location with contacts traffic comes from IPs associated with Google Internet Authority and when playing a song received from another contact Telegram connects to `.us-west-2.compute.amazonaws.com`. [58]

A 2017 investigation into Telegram on a virtualized Android device unearthed significant forensic data, determined the structure and format of these artifacts, and implemented the corresponding decoding procedures in a Java program and mapped the data stored by Telegram Messenger to the user actions that generated it. Using the above mapping, they also show how to recover the account used with Telegram Messenger, and how to reconstruct the contact list of the user, the chronology and contents of both textual and non-textual messages, and the log of the voice calls done or received by the user. It is also offering to serve as a detailed template for similar forensic analyses thanks to its thorough and well readable layout. [61]

The 2018 research compared data extraction methods on Telegram, WhatsApp, Viber and WeChat across both rooted and un-rooted Android devices, uncovering the specific file storage practices of Telegram and other messaging apps. Android Debug Bridge (adb)

commands `pull` for rooted phones or `backup` for un-rooted phones was used to extract the data. Telegram files were only accessible on rooted devices with the use of `pull` command. It found out that Telegram in version 4.5.1 stored encrypted SQLite database of chat messages at `/data/data/org.telegram.messenger/files/Cache4.db`, details about account used `/data/data/org.telegram.messenger/shared prefs/userconfing.xml`, profile photo `/data/media/0/Android/data/org.telegram.messenger/cache` and copies of files sent and received `/data/media/telegram`. Paper also offers a brief description of different kinds of chats available in Telegram and used encryption. [55]

Forensic analyses of Signal is an understudied field. 2021 study offers extensive traffic analysis of the Signal on Android phone with traffic capture and filtering done by dedicated firewall. Study revealed the obscured design of Signal app and was able to identify different activities of the Signal app such as calls, text or typing indicator. [56] This study also highlights how dynamic field is the forensic analysis of Instant Messaging as many domain names and other information have already managed to change, since the publication of this study.

4 Signal analysis

The main aim of the thesis was to compare and analyse repacked version of applications. We have compared official Signal application with unofficial one called signal-cli. Official Signal for Linux `signal-desktop` uses Electron framework as the GUI, which uses Chromium browser engine in the background.

We tested version 7.0.0-1 which we got from the stable branch of Manjaro repository [62], [63].

The signal-cli is a commandline interface for the Signal messenger. The intended use of signal-cli claimed by developer is to be used on servers to notify admins of important events but there are also mutliple TUI applications available. It occupies an important niche as it is the only unofficial commandline interface for Signal messenger and the official Signal developers do not provide one.

Signal-cli is based on custom patched libsignal-service the official Signal-Android source code, which also includes Signal officials platform agnostic libsignal, which handles all the cryptography and protocols. [64]

We tested v0.13.3 of signal-cli, specifically `signal-cli-0.13.3-Linux-native.tar.gz`, which we got from the official developer's repository. [65]

In both `signal-desktop` and `signal-cli` we tested receiving and sending messages and traced TCP active connections via `connect()` syscalls, opened files via `open()` syscalls, new processes via `exec()` syscalls with custom BPF Compiler Collection (BCC) code. We compared this output in order to evaluate whether it is a legitimate one.

For `signal-desktop` we started tracing, then started signal-desktop with command `signal-desktop` and after that we received and sent messages inside the GUI.

With `signal-cli` we have traced and performed all these operations separately because the commandline interface easily allows this. In these logs we have replaced the real numbers with a randomly generated fake ones where `+421 909 104 930` is phone number of the user of signal-cli, and `+421 909 543 173` phone number of other user with whom

we interact. We first linked our device with the main Signal app on Android phone with `./signal-cli link -n +421909104930` (this also needs to be done on signal-desktop but we did not capture this process there), `./signal-cli -u +421909104930 receive` to get the list of contacts and groups from the main device, `./signal-cli -u +421909104930 send -m "Hello, how are you?" +421 909 543 173` to send the message to `+421 909 543 173` and `./signal-cli -u +421909543173 receive` to check if there is new message from `+421 948 413 685` and receive them if there are any.

When reading this analysis keep in mind that we collected this data by running a single tracer programme in the background with no noticeable effect on performance and we can easily generate them again for any other software.

4.1 BCC tool

In order to perform the analysis we had to use a suitable tracing tool. We described available options in Chapter 2 and to use eBPF with help of the BCC toolkit. BCC performs majority of its tracing in-kernel which significantly speeds up tracing, which is the main reason why we choose it.

We have taken the structure of the provided BCC Tools as the starting point but restructuring it into the main function, additional function and classes in order to increase readability and maintainability. In contrast with the tool provided by in BCC repository also traces all the system calls that we were interested in at once as it would be suboptimal having to repeat the same tracing multiple times.

The final code that we used for tracing can be found at https://github.com/IacobusKopiirefuto/anbako-BCC.

4.2 connect

According to the signal-desktop discussion on SignalComunity forum and `dns.ts` file the current production domain names that Signal should connect to are `chat.signal.org`, `storage.signal.org`, `cdsi.signal.org`, `cdn.signal.org`, `cdn2.signal.org` and

PID	COMM	DADDR	DPORT	QUERY
13646	signal-deskt	2600:9000:a61f:527c: d5eb:a431:5239:3232	443	chat.signal.org
13646	signal-deskt	2606:4700:4400::ac40:966c	443	cdn2.signal.org
13646	signal-deskt	2606:4700:4400::ac40:966c	443	cdn2.signal.org
13646	signal-deskt	2606:4700:4400::ac40:966c	443	cdn2.signal.org
13646	signal-deskt	2a00:1450:4014:80e::2013	443	storage.signal.org
13646	signal-deskt	2600:9000:a61f:527c: d5eb:a431:5239:3232	443	chat.signal.org
13646	signal-deskt	2600:9000:2611:9200: 1d:4f32:50c0:93a1	443	cdn.signal.org
13646	signal-deskt	2600:9000:a61f:527c: d5eb:a431:5239:3232	443	chat.signal.org
448	NetworkManag	2a01:4f8:c0c:51f3::1	80	ping.manjaro.org
448	NetworkManag	116.203.91.91	80	ping.manjaro.org

Table 1: Network connections for signal-desktop

`create.signal.art` and TCP port 443 and all UDP ports are used. [66], [67]

So, if Signal applications connect to some other domains they might suggest a suspicious activity.

Below we can see the log of `connect()` syscall where `PID` is process ID of the application making the connection, `COMM` is command or application name making the network request, `DADDR` is destination address, i.e., the server's IP address, `DPORT` is destination port, commonly 443 for HTTPS connections and 80 for HTTP, `QUERY` domain name queried.

The original log also includes `IP` - the IP protocol version used for the connection (6 for IPv6, 4 for IPv4) and `SADDR` source address, i.e., the local machine's IP address which we omitted for better readability.

In Table 1 we can see that official `signal-desktop` does only the expected connections. The `ping.manjaro.org` is just a standard background noise which occurs even when no application is running and is not related to the Signal app itself but to the operating system.

Beside the fact that the `signal-cli` uses IPv4 addresses instead of the IPv6, the domain names to which it connects remain the same. Interesting fact is that which is shown by

PID	COMM	DADDR	DPORT	QUERY
11234	signal-cli	13.248.212.111	443	chat.signal.org
11234	.signal.org/	13.248.212.111	443	chat.signal.org

Table 2: Network Connections for signal-cli - send message

PID	COMM	DADDR	DPORT	QUERY
11084	signal-cli	13.248.212.111	443	chat.signal.org
11084	.signal.org/	13.248.212.111	443	chat.signal.org
11084	.signal.org/	13.248.212.111	443	chat.signal.org
11084	tokio-runtim	2603:1030:7::1	443	cdsi.signal.org
11084	signal-cli	104.18.37.148	443	cdn2.signal.org
11084	pool-6-threa	3.161.119.11	443	cdn.signal.org
11084	pool-6-threa	142.251.36.147	443	storage.signal.org

Table 3: Network Connections for signal-cli - Receive all messages

Table 2 that sending requires just two connections to `chat.signal.org` while linking your desktop device with your main device, receiving all messages or just messages from a single user require the same number of connections.

4.3 open

We also tracked `open()` syscall to monitor what files it tries to access. Both signal-desktop and `signal-cli` are expected to mainly access their own confing files and various libraries as well as content in `/usr/share/` and `~/.local/share`

The `signal-desktop` tries to access document mostly connected to form their own files and configuration files `/usr/lib/signal-desktop/`, `~/.config/Signal` but most of directories are connected to visuals of the GUI such as fonts and icons `/usr/share/fonts`,

PID	COMM	DADDR	DPORT	QUERY
448	NetworkManag	116.203.91.91	80	ping.manjaro.org
448	NetworkManag	2a01:4f8:c0c:51f3::1	80	ping.manjaro.org
11413	signal-cli	76.223.92.165	443	chat.signal.org
11413	.signal.org/	76.223.92.165	443	chat.signal.org
11413	.signal.org/	76.223.92.165	443	chat.signal.org
11413	pool-6-threa	142.251.36.147	443	storage.signal.org

Table 4: Network Connections for signal-cli - Receive message from single contact

PID	COMM	DADDR	DPORT	QUERY
10958	.signal.org/	13.248.212.111	443	chat.signal.org
448	NetworkManag	116.203.91.91	80	ping.manjaro.org
448	NetworkManag	2a01:4f8:c0c:51f3::1	80	ping.manjaro.org
10958	signal-cli	76.223.92.165	443	chat.signal.org
10958	signal-cli	76.223.92.165	443	chat.signal.org
10958	signal-cli	76.223.92.165	443	chat.signal.org
10958	.signal.org/	76.223.92.165	443	chat.signal.org
10958	signal-cli	3.161.119.11	443	cdn.signal.org
10958	pool-9-threa	142.251.36.147	443	storage.signal.org

Table 5: Network Connections for signal-cli - Link

`~/.local/share/fonts/`, `~/.fontconfig`, `/.cache/fontconfig/`, , `~/.icons/`, `~/.local/share/icons`. Then there are directories such as `/usr/share/locale/`, various libraries from `/usr/lib/`. Log of syscall also shows that the application uses Chromium based Electron framework as it uses `/dev/shm/.org.chromium.Chromium.*` and `/tmp/.org.chromium.Chromium.*` and also in this version Vulkan 3D graphics API `~/.local/share/vulkan/` and `/usr/share/vulkan/` API and open standard for 3D graphics and computing. It also accesses `/etc/passwd` which could hint an enumeration attempt if it was a malicious software but given that this is the official app and no other signs of enumeration have been caught we can consider this legitimate.

The signal-cli again proofs to be a legitimate app, which is actually much easier to monitor than the signal-desktop, mainly because it does not have GUI and so produces less noise.

In the same way as signal-desktop it accesses `/etc/passwd`, `/dev/urandom`, `/etc/ld.so.cach`, `/etc/nsswitch.conf`, `/etc/resolv.conf`, `/sys/devices/system/cpu/possible`, `/proc/stat`, `/proc/self/maps`, `/proc/self/fd` `/run/systemd/machines/chat.signal.org`.

From the general directories that we listed as being accessed by `signal-desktop` only the `/usr/lib/` was accessed by `signal-cli`. Therefore, we can assume that rest of the directories accessed by signal-desktop were required by the Electron framework GUI and were not needed for the core Signal functionality. The `signal-cli` also accesses its own local directory `~/.local/share/signal-cli` and instead of Chromium temporary directory it creates a shared object file (.so) related to SQLite JDBC driver inside temporary directory

which is likely used for Java database connectivity with SQLite. It also tries to ensure data integrity by creating a corresponding lock file to prevent multiple processes from accessing this shared object.

Overall all activity by `signal-cli` seems legitimate, it mostly accesses the same files as `signal-desktop` and even thank to been simple command line software it actually accesses much fewer files than the `signal-desktop`.

4.4 exec

The final part of the analysis is whether signal-desktop and signal-cli execute additional software only when they really need it; or they do something nefarious such as deleting files or running some commands with strange settings. Most of such actions should be already shown when tracing `open()` syscalls but this serves as a way to double check it. Furthermore, this analysis shows us in detail what arguments are used to start this commands.

In Table 6 we can see which programs are executed by signal-desktop. In `signal-desktop` (PIDs 13546, 13549, 13550) Signal Desktop application launch and child processes for zygote, where a zygote process is the one that listens for spawn requests from the main process and forks itself in response. The fact that it disables `--no-zygote-sandbox` reduces security by allowing broader access to system resources but it is not seriously concerning. The `xdg-settings` (PIDs 13566, 13582) sets the default URL scheme handlers for Signal. Manipulating URL handlers can be risky if misused by other applications, but in this context, it appears to be a legitimate configuration for Signal. Finally the `exe - /proc/self/exe` (PIDs 13616, 13646, 13679) represent various utility types being run (network service, renderer, and audio service), likely from within Signal. Here again there are flags for disabling of features like `HardwareMediaKeyHandling` and `SpareRendererForSitePerProcess`s, and flags such `--no-sandbox` and `--enable-crash-reporter` reduce security. For example disabling sandbox might be exploited via the rendering engine or enabling crash reports might leak sensitive data. [68]

Table 6: exec syscalls of signal-desktop

PID	COMM	PPID	RET	ARGS
13541	fish	1	0	"/usr/bin/fish"
13545	flatpak	13541	0	"/usr/bin/flatpak" "–installations"
13546	signal-desktop	13541	0	"/usr/bin/signal-desktop"
13549	signal-desktop	13546	0	"/usr/lib/signal-desktop/signal-desktop" "–type=zygote" "–no-zygote-sandbox"
13550	signal-desktop	13546	0	"/usr/lib/signal-desktop/signal-desktop" "–type=zygote"
13566	xdg-settings	13546	0	"/usr/bin/xdg-settings" "set" "default-url-scheme-handler" "sgnl" "signal.desktop"
13582	xdg-settings	13546	0	"/usr/bin/xdg-settings" "set" "default-url-scheme-handler" "signalcaptcha" "signal.desktop"
13616	exe	13546	0	"/proc/self/exe" "–type=utility" "–utility-sub-type=network.mojom.NetworkService" "–lang=en-US" "–service-sandbox-type=none" "–enable-crash-reporter=f7e9ea8d-135d-4de5-a67c-95010fb400d7,no_channel" "–user-data-dir=/home/jackie/.config/Signal" "–shared-files=v8_context_snapshot_data:100" "--field-trial-handle=0,i,15179410576987492959,17697197405601602943,262144" "–enable-features=kWebSQLAccess" "--disable-features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess" "–variations-seed-version"

Table 6 continued from previous page

PID	COMM	PPID	RET	ARGS
13646	exe	13546	0	"/proc/self/exe" "–type=renderer" "–enable-crash-reporter=f7e9ea8d-135d-4de5-a67c-95010fb400d7,no_channel" "–user-data-dir=/home/jackie/.config/Signal" "–app-path=/usr/lib/signal-desktop/resources/app.asar" "–no-sandbox" "–no-zygote" "–enable-blink-features=CSSPseudoDir,CSSLogical" "--disable-blink-features=Accelerated2dCanvas,AcceleratedSmallCanvases" "–first-renderer-process" "–disable-gpu-compositing" "–lang=en-US" "–num-raster-threads=2" "–enable-main-frame-before-activation" "–renderer-client-id=4" "–time-ticks-at-unix-epoch=-1715236720975010" "–launch-time-ticks=25421136890" "–shared-files=v8_context_snapshot_data:100" "--field-trial-handle=0,i,15179410576987492959,17697197405601602943,262144" "–enable-features=kWebSQLAccess" ""

Table 6 continued from previous page

PID	COMM	PPID	RET	ARGS
13679	exe	13546	0	"/proc/self/exe" "–type=utility" "–utility-sub-type=audio.mojom.AudioService" "–lang=en-US" "–service-sandbox-type=none" "–enable-crash-reporter=f7e9ea8d-135d-4de5-a67c-95010fb400d7,no_channel" "–user-data-dir=/home/jackie/.config/Signal" "–shared-files=v8_context_snapshot_data:100" "--field-trial-handle=0,i,15179410576987492959,17697197405601602943,262144" "–enable-features=kWebSQLAccess" "--disable-features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess" "–variations-seed-version"

Table 7: exec() syscalls for signal-cli send

PCOMM	PID	PPID	RET	ARGS
signal-cli	11234	10307	0	"./signal-cli -u +421909104930 send -m Ḧello, how are you?ẗ+421909543173"
sh	11237	11234	0	"/usr/bin/sh -c stty -a < /dev/tty"
stty	11239	11237	0	"/usr/bin/stty -a"
sh	11240	11234	0	"/usr/bin/sh -c stty -a < /dev/tty"
stty	11241	11240	0	"/usr/bin/stty -a"
uname	11242	11234	0	"/usr/bin/uname -o"

As we can see in Table 7 the `signal-cli` does fewer and much less complex steps than `signal-desktop`, the same as with the `connect()` and `open()` syscalls. It initiates a shell `sh` (PIDs 11237, 11240) to execute further commands. Then it executes `stty` (PIDs 11239, 11241), `stty -a` to be precise, which gets all settings for the current terminal. It uses

`uname` (PID 11242), `uname -o`, which gets the name of the operating system. As all actions end here, we can assume that these operations are done just to use correct setting for given operating system. For linking devices and receiving messages the same commands are executed.

Conclusion

The primary goal of the thesis was to conduct a comparative analysis between the official Signal application and its unofficial command-line counterpart, signal-cli. This study focused on their behavior concerning system interactions captured through syscalls while performing basic operations like sending and receiving messages.

We looked at the repacked attack and the way it can be detected through black-box analysis by using observability tools already available in the Linux environment.

We attempted the deep-dive review of these observability tools, explored their mechanism in order to evaluate strengths as well drawback of each tool. We reached conclusion that the eBPF - the in-kernel extend Berkeley Packet Filter, originally used only the `tcpdump` for packet filtering but now with overgrowing uses from sandboxing in browsers to tracing of almost any part of the OS - is the best choice. Not only because it allows us to use single tool to do all the tracing but also because it does all its tracing in kernel and sends final result to the user space which speeds up tracing significantly and makes running of software while tracing it virtually as fast as running it without tracing, unlike tool like `strace` which slow programs sometimes to the point of failure.

Deciding to use eBPF, we have used the BCC toolkit for BPF-based Linux IO analysis, networking, monitoring to create our tracing script available at https://github.com/Iacob usKopiirefuto/anbako-BCC.

Finally, we applied all this knowledge in order to find whether only application which allows using Signal Messenger via commandline interface, the signal-cli, only performs the intended function of sending and receiving Signal messages or does something at least suspicious. Our analysis have proven not only that siganl-cli but thank to having no GUI it has also does fewer system call making monitoring of its activity much easier.

With our BCC tool we explored the differences between the official Signal application, which operates on the Electron framework incorporating the Chromium browser engine, and its unofficial command-line counterpart, signal-cli, which utilizes a custom patched version of libsignal-service from the Signal-Android source code. The official Signal desktop

version tested was 7.0.0-1 sourced from the stable Manjaro repository, while the signal-cli was evaluated at version 0.13.3 from the official developer's repository. The analysis involved tracing system interactions through syscalls such as connect, open, and exec using custom BPF Compiler Collection (BCC) code to assess each application's operations and system interactions. Signal-desktop displayed more complex, interconnected operations due to its GUI-based framework, whereas signal-cli facilitated more straightforward, discrete command-line operations, allowing for more granular analysis. Both applications were confirmed to only communicate with official Signal service endpoints over secure connections, adhering to expected operational standards without any indication of malicious activities.

The analysis confirms that both the official Signal desktop application and the unofficial signal-cli adhere to expected operational standards and communicate securely with designated servers. Signal-cli provides a robust alternative for command-line based interactions with Signal services, particularly valuable for automated or backend processes on servers. The thorough syscall tracing and analysis reinforce the integrity of both applications, with no significant security concerns detected under normal usage conditions.

This research underscores the importance of syscall monitoring as a tool for validating application behavior, particularly in verifying that networked applications like Signal engage only with intended services and perform expected actions without undue exposure to the underlying system.

Future recommendations include continuous monitoring of syscall patterns with updates and patches, especially for applications like Signal-desktop that rely on complex frameworks such as Electron, which are frequently updated to address security vulnerabilities.

We can state that the main aim of our diploma thesis has been achieved.

References

[1] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy*, 2012, pp. 95–109. doi: 10.1109/SP.2012.16[1]

[2] H. Rafiq, N. Aslam, M. Aleem, B. Issac, and R. H. Randhawa, "AndroMalPack: Enhancing the ML-based malware classification by detection and removal of repacked apps for android systems," *Scientific Reports*, vol. 12, no. 1, p. 19534, Nov. 2022, doi: 10.1038/s41598-022-23766-w[2]. Available: https://doi.org/10.1038/s41598-022-23766-w

[3] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "AdRob: Examining the landscape and impact of android application plagiarism," in *Proceeding of the 11th annual international conference on mobile systems, applications, and services*, in MobiSys '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 431–444. doi: 10.1145/2462456.2464461[3]. Available: https://doi.org/10.1145/2462456.2464461

[4] L. Štefanko and P. Strýček, "Not-so-private messaging: Trojanized WhatsApp and Telegram apps go after cryptocurrency wallets." https://www.welivesecurity.com/2023/03/16/not-so-private-messaging-trojanized-whatsapp-telegram-cryptocurrency-wallets/, Mar. 16, 2023.

[5] Android Open Source Project, "Privacy changes in Android 10." https://developer.android.com/about/versions/10/privacy/changes#clipboard-data, 2023.

[6] D. Kalinin, "Analysis of a spy module inside a WhatsApp mod." https://securelist.com/spyware-whatsapp-mod/110984/, Nov. 02, 2023.

[7] L. Štefanko, "StrongPity espionage campaign targeting Android users." https://www.welivesecurity.com/2023/01/10/strongpity-espionage-campaign-targeting-android-users/, Jan. 10, 2023.

[8] MITRE, "PROMETHIUM, StrongPity, Group G0056 | MITRE ATT&CK® — attack.mitre.org." https://attack.mitre.org/versions/v14/groups/G0056/, Oct. 22, 2020.

[9] L. Štefanko, "BadBazaar espionage tool targets Android users via trojanized Signal and Telegram apps." https://www.welivesecurity.com/en/eset-research/badbazaar-espionage-tool-targets-android-users-trojanized-signal-telegram-apps/, Aug. 30, 2023.

[10] I. Golovin, "Spyware Telegram mod distributed via Google Play — securelist.com." https://securelist.com/trojanized-telegram-mod-attacking-chinese-users/110482/, Sep. 08, 2023.

[11] A. Hart, Leptopoda, and Y. Lu, "What's with the protocol using Rust?" https://community.signalusers.org/t/whats-with-the-protocol-using-rust/18957, Dec. 20, 2020.

[12] B. Gregg, "BSidesSF 2017: Security monitoring with eBPF." https://www.brendangregg.com/Slides/BSidesSF2017_BPF_security_monitoring, 2017.

[13] B. Gregg, "What is Observability." https://brendangregg.com/blog/2021-05-23/what-is-observability.html, May 23, 2021.

[14] J. Evans, "Linux tracing systems & how they fit together." https://jvns.ca/blog/2017/07/05/linux-tracing-systems/, 2017.

[15] B. Gregg, "Choosing a Linux Tracer (2015)." https://brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html, 2015.

[16] J. Edge, "Unifying kernel tracing." https://lwn.net/Articles/803347/, 2019.

[17] Terenceli, "Linux tracing - kprobe, uprobe and tracepoint." https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2020/08/05/tracing-basic, 2020.

[18] P. Kranenburg, "Strace - an alternative syscall tracer, Part01/04." https://stuff.mit.edu/afs/sipb/project/eichin/cruft/machine/sun/sun-Strace, 1992.

[19] B. Gregg, "Strace Wow Much Syscall." https://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html, May 11, 2014.

[20] E. Syromyatnikov, "FOSDEM 2020 - strace: Fight for performance." https://archive.fosdem.org/2020/schedule/event/debugging_strace_perfotmance/, Feb. 02, 2020.

[21] D. Levin, "FOSDEM 2020 - Postmodern strace." https://archive.fosdem.org/2020/schedule/event/debugging_strace_modern/, Feb. 02, 2020.

[22] J. Corbet, "Replacing ptrace()." https://lwn.net/Articles/371501/, 2010.

[23] T. Heo, "Proposal for ptrace improvements." https://lwn.net/Articles/430373/, 2011.

[24] J. Corbet, "Improving ptrace()." https://lwn.net/Articles/432114/, 2011.

[25] M. Müller, "Can a process have multiple parents? Why or why not?" https://stackoverflow.com/questions/42333659/can-a-process-have-multiple-parents-why-or-why-not, 2017.

[26] Andrew, "Bypassing ptrace in gdb." https://stackoverflow.com/questions/33646927/bypassing-ptrace-in-gdb, 2015.

[27] OffSecServices, "OffSec's Exploit Database Archive — exploit-db.com." https://www.exploit-db.com/exploits/40839, 2016.

[28] Y. Shafet, "The Race to Limit Ptrace." https://www.rezilion.com/blog/the-race-to-limit-ptrace/, 2020.

[29] J. Evans, "A zine about strace." https://jvns.ca/blog/2015/04/14/strace-zine/, 2015.

[30] S. Abdalla, "Say this five times fast: Strace, ptrace, dtrace, dtruss." https://dev.to/captainsafia/say-this-five-times-fast-strace-ptrace-dtrace-dtruss-3e1b, 2019.

[31] N. Abda, "Understanding ptrace — abda.nl." https://abda.nl/posts/understanding-ptrace/, 2019.

[32] S. McCanne, G. Combs, and L. Degioanni, "SHARKFEST '11 Keynote Address." https://sharkfestus.wireshark.org/sharkfest.11/, 2011.

[33] tshepang, "Difference between sniffer tools — networkengineering.stackexchange.com." https://networkengineering.stackexchange.com/a/10075, 2017.

[34] wireshark, "Tools - Wireshark Wiki — wiki.wireshark.org." https://wiki.wireshark.org/Tools, 2020.

[35] user862787, "Performance and efficiency comparing between dump tools: Tcpdump, tshark, dumpcap." https://stackoverflow.com/a/22234865, 2014.

[36] J. Evans, "Let's learn tcpdump!" https://wizardzines.com/zines/tcpdump/, 2017.

[37] eBPF Foundation, "eBPF Projects." https://ebpf.foundation/projects/.

[38] M. Steve, "Shark fest 2011 keynote." https://sharkfestus.wireshark.org/sharkfest.11/, 2011.

[39] S. McCann, "The BSD packet filter: A new architecture for user-level packet capture." http://www.tcpdump.org/papers/bpf-usenix93.pdf, 1992.

[40] L. Rice, "Learning eBPF — chapter 1." https://www.oreilly.com/library/view/learning-ebpf/9781098135119/ch01.html, 2023.

[41] Linux Kernel, "BPF licensing." https://www.kernel.org/doc/html/latest/bpf/bpf_licensing.html.

[42] A. Starovoitov, "BPF in-kernel virtual machine." https://netdevconf.info/0.1/sessions/15.html, 2015.

[43] E. Dumazet, "Re: [PATCH v2] net: Filter: Just in time compiler." Email to David Miller, Apr. 03, 2011. Available: https://lore.kernel.org/netdev/1301838968.2837.200.camel@edumazet-laptop/

[44] Chromium Project, "The seccomp sandbox." Online. Available: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux/sandboxing.md#The-sandbox-1

[45] A. Chapman, "Seccomp and Seccomp-BPF." https://ajxchapman.github.io/linux/2016/08/31/seccomp-and-seccomp-bpf.html, 2016.

[46] Kernel Documentation, "Seccomp BPF (SECure COMPuting with filters)." Online. Available: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

[47] Linux Kernel Developers, "Source code of xt_bpf.c in linux kernel v3.9." Online, 2013. Available: https://elixir.bootlin.com/linux/v3.9/source/net/netfilter/xt_bpf.c

[48] Linux Kernel Developers, "Source code of cls_bpf.c in linux kernel v3.13." Online, 2014. Available: https://elixir.bootlin.com/linux/v3.13/source/net/sched/cls_bpf.c

[49] Kernel Documentation, "Linux socket filtering aka berkeley packet filter (BPF)." Online, 2024. Available: https://www.kernel.org/doc/html/latest/networking/filter.html

[50] D. Borkmann, "Re: [PATCH net-next v4 0/9] BPF updates." Email to David Miller, Mar. 28, 2014. Available: https://lore.kernel.org/netdev/1396029506-16776-1-git-send-email-dborkman@redhat.com/

[51] Wikipedia contributors, "History of eBPF." Wikipedia, The Free Encyclopedia, May 2024. Available: https://en.wikipedia.org/wiki/EBPF#History

[52] Polynka, "Lua in the kernel?" https://lwn.net/Articles/831083/, 2020.

[53] B. Gregg, "eBPF observability tools are not security tools." https://brendangregg.com/blog/2023-04-28/ebpf-security-issues.html, 2023.

[54] J. Gregorio, A. Gardel, and B. Alarcos, "Forensic analysis of telegram messenger for windows phone," *Digital Investigation*, vol. 22, pp. 88–106, 2017, doi: https://doi.org/10.1016/j.diin.2017.07.004. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301032

[55] K. Rathi, U. Karabiyik, T. Aderibigbe, and H. Chi, "Forensic analysis of encrypted instant messaging applications on android," in *2018 6th international symposium on digital forensic and security (ISDFS)*, 2018, pp. 1–6. doi: 10.1109/ISDFS.2018.8355344[4]

[56] A. Afzal, M. Hussain, S. Saleem, M. K. Shahzad, A. T. S. Ho, and K.-H. Jung, "Encrypted network traffic analysis of secure instant messaging application: A case study of signal messenger app," *Applied Sciences*, vol. 11, no. 17, 2021, doi: 10.3390/app11177789[5]. Available: https://www.mdpi.com/2076-3417/11/17/7789

[57] K. Gupta, D. Oladimeji, C. Varol, A. Rasheed, and N. Shahshidhar, "A comprehensive survey on artifact recovery from social media platforms: Approaches and future research directions," *Information*, vol. 14, no. 12, 2023, doi: 10.3390/info14120629[6]. Available: https://www.mdpi.com/2078-2489/14/12/629

[58] M. N. Yusoff, A. Dehghantanha, and R. Mahmod, "Chapter 5 - network traffic forensics on firefox mobile OS: Facebook, twitter, and telegram as case studies," in *Contemporary digital forensic investigations of cloud and mobile applications*, K.-K. R. Choo and A. Dehghantanha, Eds., Syngress, 2017, pp. 63–78. doi: https://doi.org/10.1016/B978-0-12-805303-4.00005-8. Available: https://www.sciencedirect.com/science/article/pii/B9780128053034000058

[59] "Windows 10 Mobile End of Support: FAQ." Microsoft, 2019. Available: https://support.microsoft.com/en-us/windows/windows-10-mobile-end-of-support-faq-8c2dd1cf-a571-00f0-0881-bb83926d05c5

[60] C. Reilly, "Goodbye, Windows 10 Mobile, tweets Joe Belfiore — cnet.com." https://www.cnet.com/tech/mobile/windows-10-mobile-features-hardware-death-sentence-microsoft/, 2017.

[61] C. Anglano, M. Canonico, and M. Guazzone, "Forensic analysis of telegram messenger on android smartphones," *Digital Investigation*, vol. 23, pp. 31–49, 2017, doi: https://doi.org/10.1016/j.diin.2017.09.002. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301767

[62] kpcyrd, "Arch Linux - signal-desktop 7.8.0-1 (x86_64) — archlinux.org." https://archlinux.org/packages/extra/x86_64/signal-desktop/, 2024.

[63] signal, "Signalapp/Signal-Desktop." https://github.com/signalapp/Signal-Desktop, 2024.

[64] Signal, "Signal-Android/libsignal-service." https://github.com/signalapp/Signal-Android/tree/main/libsignal-service, 2023.

[65] AsamK, "Release v0.13.3 · AsamK/signal-cli." https://github.com/AsamK/signal-cli/releases/tag/v0.13.3, 2024.

[66] Herohtar, "URLs for Prioritizing on my firewall." https://community.signalusers.org/t/urls-for-prioritizing-on-my-firewall/38553/4, 2023.

[67] Signal, "Firewall and Internet settings." https://support.signal.org/hc/en-us/articles/360007320291-Firewall-and-Internet-settings.

[68] Chromium, “Chromium Docs - docs/linux/zygote.md.” https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux/zygote.md.