Monitoring and protecting SSH sessions with eBPF

Guillaume Fournier
gui774ume.fournier@gmail.com

Datadog

Abstract. According to Verizon's Data Breach Investigations Report [24], credential theft, user errors and social engineering account for 67% of the data breaches that occurred in 2020. This is not a particularly new problem: credentials in general have always been a sensitive issue, particularly when users have to interact with them. Even with the best security recommendations in place, like credentials rotation or certificate based authentication, SSH access will always be a security hazard. This article intends to explain why the default Mandatory Access Controls of Linux are not enough to provide a granular control over SSH sessions. eBPF will be explored as a potential solution to provide a refined access control granularity along with SSH sessions security audit capabilities.

1 Introduction

Secure Shell (SSH) [11] is a network protocol that provides users with a secure way to access a host over an unsecure network. Multiple actors in an organisation usually require this access:

— Developers use it to debug applications in a staging or sometimes production environment.

— Software Reliability Engineering (SRE) teams and system administrators often perform maintenance and system configuration tasks over SSH.

— Security administrators often require access to machines in production environments to perform security investigations and incident response.

In theory, the principle of least privileges should be applied, and each actor should be granted the bare minimum access required to perform its tasks. Although SRE teams and security engineers are likely to require privileged access, developers might not always need it. Regardless of the level of access on a machine, developers should also have access only to the hosts that run the services they work on.

Unfortunately, those principles are often hard to follow. First, debugging an application often means using runtime performance monitoring tools to understand why a service is not performing as expected. For example, you might want to analyse the ext4 operations latency distribution using a tool like *ext4dist* [12]. Some of those performance monitoring tools require root access, which means that many developers will eventually request permanent root access. Moreover, with the growing adoption of container orchestration tools like Kubernetes, hosts are no longer dedicated to specific services.[1] Many developers will eventually require root access to some of the nodes of the infrastructure,[2] thus also granting them access to some pods of services they do not own. As companies grow and engineering teams expand, the number of privileged users on the infrastructure skyrockets, making it particularly hard for the security team to monitor SSH sessions and contain the blast radius of leaked credentials.

This paper explores how eBPF can provide a solution to monitor SSH sessions, while providing a security team with a new access control layer. This new access control grants temporary access to scoped resources. In other words, the “all or nothing” access usually granted to Linux users no longer applies: a *sudoer* user might be able to become root, its access will still be restricted to the access granted to the SSH session.

2 Securing SSH sessions: state of the art and limitations

2.1 Security recommendations to protect SSH access

It is generally agreed that public key authentication for SSH is better than using passwords [4]. Unfortunately, even public keys are far from being perfect because they come with the burden of key management. The main criticism is usually that key management is complex and does not scale well. From key rotation to keeping up with the employees turnover, updating the authorized keys of hundreds of hosts can quickly become a logistical nightmare. This is why the current security recommendation is to switch to a certificate based authentication. Instead of using keys with permanent access, temporary certificates are delivered by a Certificate Authority (CA) so that users can authenticate to hosts and hosts to users. Then, logging into a host becomes simply the process of showing each other a certificate and validating that it was signed by the trusted CA.

1. Kubernetes can be configured to dedicate some hosts to specific workloads, but this requires a custom setup that often voids the entire point of a Kubernetes environment. In this document, we expect Kubernetes to follow its default configuration, which is to allow any workload to be scheduled on any host of a cluster.

2. access to a pod through the *kubectl exec* command is rarely enough to debug a service in Kubernetes.

The second security recommendation is to enforce the use of a bastion host (sometimes also called a jumpbox). In a few words, a bastion host is a server that is specifically designed to be the only gateway for SSH access into an infrastructure. A bastion host helps reduce the points of entry into an infrastructure, while also providing a single place to monitor and audit SSH access. From the hosts side of things, you should also configure the network firewall to block incoming SSH connections that are not from the bastion host.

Next is Multi-factor authentication (MFA). MFA or 2-factor authentication makes it harder for attackers to log into your infrastructure by enforcing the need for two different login methods for the authentication to be successful. Multiple options exist: hardware authentication devices (like a Google Titan or a Yubikey), text messages, One-time Password (OTP) applications (like Google Authenticator or Duo), etc. Ubuntu published a comprehensive guide to set up Google Authenticator with OpenSSH [23].

The final security recommendation is to audit login logs of the SSH server. A login audit trail will be particularly useful during an investigation to gather the list of hosts accessed by a compromised user.

2.2 Why aren't those security recommendations enough ?

MFA isn't perfect, stolen credentials are still a threat One of the most important goals of the recommendations described in the previous section, is to make sure that access is granted temporarily (certificates should be configured to last a few hours, and MFA authentication is time sensitive by design) and harder to compromise (stealing credentials is not enough, attackers need to compromise your MFA method too). Although this mitigates the probability of being compromised, it does not affect the impact of stolen credentials. Indeed, even MFA solutions can be compromised. The most famous and recent example of 2-factor authentication abuse is what happened to Twitter's CEO Jack Dorsey in 2019 [1]. In a few words, Jack Dorsey was the target of a SIM swap attack [18] which allowed an attacker to bypass an SMS based MFA. Ultimately, the attacker was able to control Jack Dorsey's Twitter account. Some might argue that the problem is SMS based MFA, because it relies on the mobile carrier to carefully protect your phone number and account. This is why authenticator applications or hardware authentication devices are usually recommended for implementing MFA.

Unfortunately, MFA is hard to get right, regardless of the intermediaries (like a phone carrier) that might be involved in the process. Proofpoint [20] recently discovered critical vulnerabilities in cloud environments where

WS-Trust [19] is enabled. They were reportedly capable of bypassing MFA for multiple Identity Providers and cloud applications that used the protocol, such as Microsoft 365. Hardware authentication devices are no exception to the rule, multiple vulnerabilities were discovered over the past few years that could be exploited to either extract the encryption key [17] of the hardware, or bypass the verification entirely [9].

Regardless of all those hiccups, MFA is still a must have to protect the SSH access to an infrastructure. However, it is important to note that it is a security layer and not a bullet proof strategy. Regardless of MFA design flaws, human errors and phishing attacks are still omnipresent, which means that even a well configured infrastructure is exposed to the threat of stolen credentials and malicious access. In other words, MFA makes it harder for remote access to be compromised, but we are still missing a security layer to mitigate the impact at runtime of stolen credentials.

SSH access is rarely granular This subject has already been discussed in the introduction of this article, but in a few words, it is likely that many developers have an unnecessarily high level of access by default. Whether it is to enable debugging and monitoring tools, or because a containerized infrastructure blurs the lines between services, the blast radius of stolen credentials is often really high. The data breach that affected GoDaddy in 2019 is probably the best example showing how quickly things can escalate when SSH credentials with a high level of access are stolen [25]. In short, an unauthorized individual gained access to login credentials that eventually led to the compromise of 28 000 customers.

To be fair, Linux doesn't make it particularly easy to configure granular access at scale. Kernel capabilities [13] and various setuid or setgid tricks could be used to avoid granting sudo access to a developer who needs to run various debugging and monitoring tools. Unfortunately, using them requires a special setup on each machine and having to redeploy an updated configuration through a tool like Chef [2] or Ansible [22] to an entire infrastructure takes time and simply doesn't scale. Eventually, developers will request temporary or permanent sudo access, thus forcing the security team into another logistical nightmare.

Regardless of the sudo access problem, developers will require legitimate access to sensitive resources such as databases or applications credentials. Even when it is justified, accessing this data should be carefully monitored, and if possible, should require an additional layer of audit and access control.

SSH sessions audit logs are limited to logins and logouts Audit logs are the last pain points of SSH sessions. Most SSH servers will export login and logout events, but this is not enough to understand what a user did on the host. Collecting the shell history could be an option, but attackers are likely to clean up their tracks behind them, so it cannot be considered as a reliable solution. Security teams will have to rely on additional runtime security monitoring tools to monitor sensitive processes and file system activity.

3 SSH sessions monitoring and protecting with eBPF

ssh-probe [8] is an open source utility powered by eBPF that aims at monitoring and protecting SSH sessions at runtime. First, we'll talk about the live session monitoring feature of *ssh-probe*, showing how eBPF can be used to monitor SSH sessions in real time. Then, we'll deep dive into the session based, scope based and time based access control implemented by *ssh-probe*. This project was developed for OpenSSH because it is open source and one of the most popular SSH implementations.

3.1 Live session monitoring

eBPF is a well known technology used for tracing kernel level activity [10]. Multiple eBPF program types exist [5]: some are dedicated to network use cases, others kernel tracing (like Kprobes [14] or Tracepoints [15]), etc. One of them is dedicated to tracing user space processes and more specifically user space function calls with exported symbols. In a few words, this means that you can execute an eBPF program any time an exported symbol of a user space binary is called. This hooking mechanism is called a Uprobe [16], and the context provided to Uprobe programs contains the arguments given to the user space function.

In our case, the OpenSSH daemon exports a symbol called *setlogin* which is called when a new session is created with the username associated with the session. This means that by hooking a Uprobe on the *setlogin* symbol of OpenSSH, we can detect the first process of a new SSH session and associate it to its rightful user. We decided to use this hook point to detect the creation of a new session because it was a practical way to access the username directly. However, note that many other kernel space events could have been chosen too. For example, we could have decided to detect when *sshd* calls the *setuid* syscall to set the user ID of the SSH session.

Once a new session is detected, our eBPF programs create a random session ID that will be used to track the new session. More precisely, *ssh-probe* uses multiple Kprobes to track processes lifecycle (such as *fork* events, *execve* events, *exit* events, etc) and make sure that the session ID is inherited from a parent PID to its children. Since this information is stored in an eBPF hashmap [5] indexed by pid, we can match any kind of kernel level activity back to its SSH session. *ssh-probe* uses up to 127 hook points in the kernel to track:

- Process scheduling events: *ssh-probe* can report the list of processes that were executed by an SSH session.
- Sensitive process operations: *ssh-probe* looks for known process injection techniques that could be used to alter production services or access sensitive data.
- Process credentials update: *ssh-probe* can report when a process changes its user, group or kernel capabilities. This is important to understand what access a user had during a session.
- File system activity: *ssh-probe* can detect when sensitive files are accessed. Since we couldn't possibly send the exhaustive list of all file system events, you'll need to provide a list of file patterns that should generate an audit log.
- Socket creation: An accurate tracking of network activity wasn't in the scope of this project. This is why we decided to monitor socket creations to notify that an SSH session generated some network traffic. If you're interested in monitoring and protecting network activity at the process level, we published an article on that topic at SSTIC 2020 [6].
- Sensitive kernel events: *ssh-probe* looks for sensitive kernel level activity like the insertion of a new kernel module, system clock adjustments, etc.

Those events are then sent back to user space using an eBPF perf map [5], so that *ssh-probe* can forward them to a log ingestion backend like Datadog. Then, Datadog provides the ability to visualize and regroup the generated events by session, allowing the user to reconstruct SSH sessions remotely and in real time.

3.2 Session, scope and time based access control

Apart from monitoring SSH sessions, *ssh-probe* is also capable of enforcing security profiles. A security profile defines, for a given user, what may or may not happen during an SSH session. For example, profiles usually include a list of sensitive file patterns, a list of binaries and the default behavior that *ssh-probe* should enforce for predefined lists of syscalls. In short, *ssh-probe* is capable of enforcing access to the resources listed in the previous section. You can find an example profile in the code repository of the project [7]. Each entry of the profile can be configured so that *ssh-probe* takes one of the following actions:

— *allow*: *ssh-probe* grants access to the ressource and generates an audit log.
— *block*: *ssh-probe* denies access to the ressource. Most of the time, this translates into blocking a syscall and overriding its answer to "permission denied" with the *bpf_override_return* helper [3].
— *MFA*: until an MFA authentication is successful, *ssh-probe* denies access to the ressource just like the *block* action. The MFA authentication mechanism is described below.
— *kill*: *ssh-probe* kills the SSH session.

One interesting aspect of this mechanism is that it is implemented on top of the normal access controls of Linux. In other words, it cannot be used to grant more access than what is already given to the Linux user. This design is particularly useful when you want to restrict the access given to a *sudoer*. Even if the user elevates its privileges to the root user, *ssh-probe* will still be able to restrict its access to predefined resources such as specific binaries (namely the tracing and debugging tools we talked about in the previous sections). Similarly, sensitive files can be blocked even from the root user. For example, there is no reason for a *sudoer* developer to ever access a file in "/root/.ssh/*". Similarly, you might want to audit accesses to credential files like "/etc/shadow".

Another important feature of *ssh-probe* is the ability to grant temporary access to specific resources based on an additional MFA verification. We implemented a scope and time based MFA verification that grants access only to the active SSH session. In other words, 2 sessions that originated from the same user may not have the same level of access at a given time.

On a technical standpoint, we had an interesting challenge to solve: how can a user provide its MFA token to *ssh-probe* without using a socket ? Indeed, you might have noticed that one of the sections available in a profile is *socket_creation*. This section controls if the processes of an SSH session are allowed to create a socket. If the MFA verification required the use of a socket to communicate with *ssh-probe*, then sockets would have had to be allowed by default. So, we eventually came up with the idea of overloading the *stat* syscall using eBPF, so that a user space program can send data to *ssh-probe* without having to connect to any local endpoint. Moreover, the session doesn't have to identify itself to *ssh-probe*, since our eBPF programs already track the active session in kernel space. In other words, in order to request temporary access to a specific resource, a session simply needs to call the *stat* syscall with a valid one-time password (OTP). We provided a utility (called *ssh-probe-auth*) to facilitate this authentication. Figure 1 explains in more details how this mechanism works.

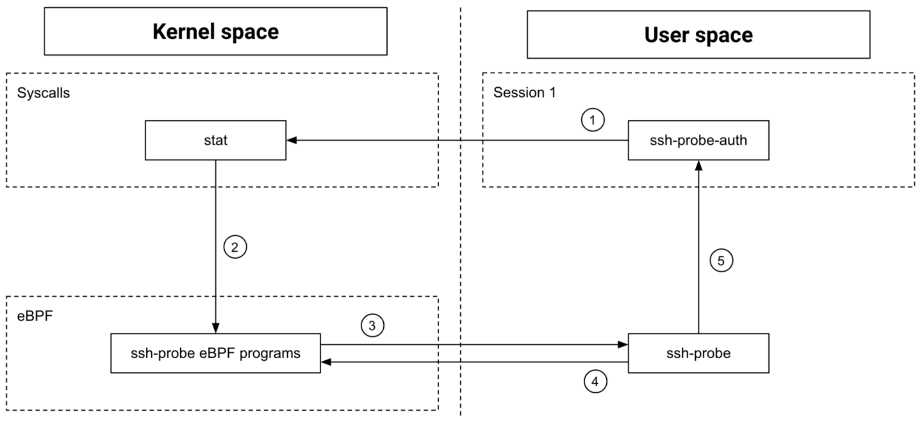

Fig. 1. MFA implementation with eBPF

1. *ssh-probe-auth* calls the *stat* syscall with the following input parameter:

```
stat("otp://fim:10000@234123")
```

Listing 1. MFA *stat* syscall

   — *fim* is the scope of the request
   — *10000* is the duration of validity of the request
   — *234123* is the one-time password (OTP)
2. *ssh-probe* placed a kprobe on the *stat* syscall, which means that one of our eBPF programs will be called with the input parameters of the syscall before the syscall is actually executed.
3. The eBPF program on *stat* parses the request and forwards it to *ssh-probe* in user space using an eBPF perf map [5]. The session credentials are appended to the request.
4. *ssh-probe* verifies the provided OTP for the active session and, if access is granted, pushes a temporary token in an eBPF map to let

our access control know that access should be temporarily granted for the given resource and session.

5. *ssh-probe* sends a signal to *ssh-probe-auth* with the outcome of the MFA verification. SIGUSR1 is used to notify that access is granted for the requested resource and duration of time, SIGUSR2 is used to notify that access is denied.

3.3 Limitations and future work

This project is still under development and we want to add more features. For example, profiles could be even more fine grained and define access on a per user and process basis. For example, you might want to allow read access to "/etc/passwd" to the *sudo* binary, but there is no reason for a developer to access this file with *cat*. Similarly, you might want to grant network access to your various network tracing tools, but not to *bash*.

Moreover, our access control is session based, and sessions are tracked using the processes lineage at runtime. Any mechanism that can trigger the execution of a binary outside of the process tree of the session would not be subject to the session access control. For example, starting a container would create a child process of the container daemon. Similarly, a *cron* job or a *systemd* service would be excluded from the SSH session. For now, blocking specific binaries like the docker client and specific files like "/etc/crontab" can be used as a temporary workaround.

Another important limitation of *ssh-probe* is that it uses the *bpf_override_return* helper [3] to block syscalls. Unfortunately, this helper is only available if the kernel was built with "CONFIG_BPF_KPROBE_OVERRIDE=y". Although it is present on some Linux distributions like Ubuntu Focal, this cannot be considered as a universal solution. Fortunately, new eBPF features are constantly added in the Linux Kernel, including some security enforcement capabilities like the Kernel Runtime Security Instrumentation (KRSI) [21]. In a few words, KRSI can be used to implement a Linux Security Module with eBPF, thus introducing the ability to reliably enforce *ssh-probe*'s security profiles.

4 Conclusion

SSH access is one of those services that can brutally undermine any security measure you might have put in place to protect your infrastructure. Its security is of critical importance for a company and the data of its customers. Although the recommended best practices go a long way towards reducing the risk for an SSH access to be compromised, they do not really impact the blast radius of stolen credentials. *ssh-probe* shows how eBPF can be leveraged to provide reliable visibility into active SSH sessions in real-time, while providing an additional session, scope and time based access control.

References

1. Brian Barrett. How Twitter CEO Jack Dorsey's Account Was Hacked. `https://www.wired.com/story/jack-dorsey-twitter-hacked`, 2020.
2. Chef. Chef Infrastructure Automation. `https://www.chef.io/products/chef-infra`.
3. Jonathan Corbet. BPF-based error injection for the kernel. `https://lwn.net/Articles/740146/`, 2017.
4. Marlon Dutra. Scalable and secure access with SSH. `https://engineering.fb.com/2016/09/12/security/scalable-and-secure-access-with-ssh`, 2016.
5. Lorenzo Fontana and David Calavera. Linux Observability with BPF. November 2019.
6. Guillaume Fournier. Process level network security monitoring and enforcement with eBPF. *SSTIC*, 2020.
7. Guillaume Fournier. ssh-probe profile example. `https://github.com/Gui774ume/ssh-probe/blob/c7364d7eef0a76bc67e35ff3d85768467862a99d/profiles/vagrant.yaml`, 2021.
8. Guillaume Fournier. ssh-probe source code. `https://github.com/Gui774ume/ssh-probe`, 2021.
9. Andy Greenberg. Chrome lets hackers phish even "Unphishable" Yubikey users. `https://www.wired.com/story/chrome-yubikey-phishing-webusb`, 2018.
10. Brendan Gregg. BPF Performance Tools: Linux System and Application Observability. December 2019.
11. IETF. SSH protocol architecture. `https://tools.ietf.org/html/rfc4251`.
12. IOVisor. Summarize ext4 operation latency distribution as a histogram. `https://github.com/iovisor/bcc/blob/master/tools/ext4dist.py`.
13. Linux. Kernel capabilities man pages. `https://man7.org/linux/man-pages/man7/capabilities.7.html`.
14. Linux. Kprobe documentation. `https://www.kernel.org/doc/Documentation/kprobes.txt`.
15. Linux. Tracepoint Documentation. `https://www.kernel.org/doc/html/latest/trace/tracepoints.html`.
16. Linux. Uprobe Documentation. `https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt`.
17. Victor Lomne and Thomas Roche. A side journey to Titan. `https://ninjalab.io/wp-content/uploads/2021/01/a_side_journey_to_titan.pdf`, 2021.

18. MITRE. Sim card swap. https://attack.mitre.org/techniques/T1451.
19. OASIS. WS-Trust 1.4. http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html, 2012.
20. Or Safran and Itir Clarke. New Vulnerabilities Bypass Multi-Factor Authentication for Microsoft 365. https://www.proofpoint.com/us/blog/cloud-security/new-vulnerabilities-bypass-multi-factor-authentication-microsoft-365, 2020.
21. KP Singh. Kernel Runtime Security Instrumentation. https://lwn.net/Articles/798918, 2019.
22. Red Hat Software. Ansible IT automation. https://github.com/ansible/ansible.
23. Ubuntu. Configure SSH to use two-factor authentication. https://ubuntu.com/tutorials/configure-ssh-2fa.
24. Verizon. Data breach investigations report (DBIR). https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf, 2020.
25. Davey Winder. GoDaddy Confirms Data Breach: What Customers Need To Know. https://www.forbes.com/sites/daveywinder/2020/05/05/godaddy-confirms-data-breach-what-19-million-customers-need-to-know, 2020.