Fingerprinting Tooling used for SSH Compromisation Attempts

Vincent Ghiëtte, Harm Griffioen, and Christian Doerr
TU Delft, Cyber Threat Intelligence Lab
{v.d.h.ghiette, h.j.griffioen, c.doerr}@tudelft.nl

Abstract

In SSH brute forcing attacks, adversaries try a lot of different username and password combinations in order to compromise a system. As such activities are easily recognizable in log files, sophisticated adversaries distribute brute forcing attacks over a large number of origins. Effectively finding such distributed campaigns proves however to be a difficult problem.

In practice, when adversaries would spread out brute-forcing over multiple sources, they would likely reuse the same kind of software across all of these origins to simplify their operation and reduce cost. This means if we are able to identify the tooling used in these attempts, we could cluster similar tool usage into likely collaborating hosts and thus campaigns. In this paper, we demonstrate that it is possible to utilize cipher suites and SSH version strings to generate a unique fingerprint for a brute-forcing tool used by the attacker.

Based on a study using a large honeynet with over 4,500 hosts, which received approximately 35 million compromisation attempts over the period of one month, we are able to identify 49 tools from the collected data, which correspond to off-the-shelf tools, as well as custom implementations. The method is also able to fingerprint individual versions of tools, and by revealing mismatches between advertised and actually implemented features detect hosts that spoof identifying information. Based on the generated fingerprints, we are able to correlate login credentials to distinguish distributed campaigns. We uncovered specific adversarial behaviors, tactics and procedures, frequently exhibiting clear timing patterns and tight coordination.

1 Introduction

Secure Shell (SSH) is a widely used protocol to operate services on a remote host over a network. One of the commonly used services of SSH is remote terminal access, which allows a user to execute programs on a remote system. The protocol authenticates a user based on a public key or an username/password combination, which prohibits malicious users to connect and exploit the host.

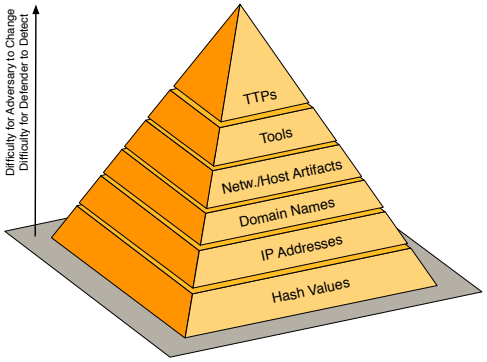

Figure 1: While basic Indicators of Compromise (IoC) are easy to gather and distribute, they are trivially changed by an adversary. For effective, more persistent detection it is necessary to assemble threat intelligence that covers behavioral features of the attacker. [3]

Due to the extensive use of the protocol, SSH is a popular target in brute forcing attacks. While system administrators are able to change the usernames and passwords used by the device, a lot of devices are still configured to use standard username and password combinations. As many devices are left with default configurations, simply trying a list of common username and password combinations proves effective enough for attackers to massively scan for, and attack SSH devices using this method.

While unsophisticated attackers would run through an extensive username/password candidate list in order to gain access, such behavior would be quickly visible in log files, and source addresses with repeated failed attempts are routinely blocked by intrusion detection systems (IDS) or monitoring systems such as fail2ban. Advanced adversaries would thus split the brute forcing out over multiple hosts, but in order to simplify the administration, usage and lower the cost, they would typically run a similar software across systems.

Current detection revolves mostly around simple indicators to detect malicious behavior. Virus scanners or intrusion detection systems for example rely on signatures and hashes to identify malicious activity, and also the IP addresses of malicious hosts scanning and brute forcing logins is enumerated and distributed in IP block lists. As explained by the so-called "pyramid of pain" [3] depicted in figure 1, these indicators are however trivially changed for an adversary, for example by simply recompiling a malware or moving the activity to a newly compromised host or proxy. In result, such information is unsuited to stop adversaries for long and at a broader scale. An alternative is the detection based on complex indicators, such as the systems or tools used or the tactics in a compromise, as they are much more difficult and costly for an adversary to change. If we can detect a particular software or modus operandi used for brute forcing SSH, we can reliably identify malicious activity, regardless of the IP address it is coming from and whether this address was participating in such activities before.

In this paper, we introduce methods for fingerprinting software stacks and tools used in SSH connections. This will help to study and follow the activities of adversaries, as an attacker will most likely distribute the same tool over a number of hosts to leverage the economies of scale. By detecting attacks by their used tools, attackers will have to change their software between campaigns, and even between different hosts. This greatly increases the cost for attackers, and can price them out of the system.

Our approach extracts session negotiation information such as the list and ordering of key exchange algorithms, cipher suites, or compression algorithms which are exchanged in clear text during the SSH session initiation. This means that using this approach, we do not need to interfere with the connection itself, meaning that the method is completely passive, and as the fingerprint is derived from the SSH handshake, we are able to identify brute forcing attempts even before the first password is sent to the system.

This paper makes the following contributions:

- We introduce the concept of fingerprinting to the SSH protocol and demonstrate based on a large corpus of 35 million brute forcing attempts that fingerprinting is suited to identify tools that are used by adversaries. By detecting attacks on this level, the cost for adversaries rises as they need to build new tools for every campaign.

- We deploy the technique to 4,500 honeypots with the aim of gaining cyber threat intelligence about the practices of adversaries. We empirically show the presence of 49 different tools, and show that a cluster of hosts relies on the same toolchains. We furthermore find evidence of large, distributed campaigns of collaborating hosts.

The remainder of this paper is structured as follows: Section 2 describes the state of the art in fingerprinting and SSH brute forcing. Section 3 provides an overview of the SSH protocol and components necessary to introduce the proposed method. Section 4 explains the fingerprinting methodology. Section 5 provides details about the design and scale of our honeynet. The evaluation of our proposed method is presented in Section 6. Using our method, we find a large number of actors, each featuring different strategies, tactics and resources. Finally, Section 7 summarizes and concludes our work.

2 Related Work

As stated in the previous section, a sustainable cyber defense best focuses not on identifiers of specific malicious instances, but on characteristics that are constant over multiple instances. One way of generating these characteristics is to fingerprint the tools used by attackers. Our main claim in this paper is that we can extract fingerprints from the SSH connection negotiation that can be used to distinguish different tools. Two lines of related work are important to class the proposed work, first previous research on fingerprinting protocols, and second previous research in brute force detection.

First, while fingerprinting has not been done in SSH, differences in cipher suite strings have been used in the SSL/TLS protocol suite to identify server or client software. To fingerprint clients, Husak et al. [9] were able to infer the used client application based on the cipher suites that were used in the connection. The authors found that many applications support different cipher suites for establishing a connection, and some applications also send the cipher suites in a different order. Therefore, the authors were able to fingerprint client applications using only the cipher suites presented in the handshake. Durumeric et al. [4] applied the analysis of advertised clients (through the HTTP User-Agent) and implemented SSL/TLS handshakes to detect the nature of the client connecting, and thereby identify middleboxes that intercepted the TLS connection between client and server. Fingerprinting specific implementations is also possible by detecting specific patterns in which header fields [5] or packet payloads [6] are set and encapsulated in scan and attack traffic.

Fingerprinting the traffic sent through encrypted channels has been done by Sun et al. [18]. Their algorithm is able to identify which webpages are visited from the amount of traffic sent during the page load. Similar research by Korczynski et al. [11] uses Markov chains to generate fingerprints for different services based on the SSL session. Their research shows that they are able to fingerprint certain applications with a high confidence level. In the case of SSL, research has focused on fingerprinting clients and client behavior. Our SSH fingerprinting method leverages the same intuitions, but is tailored towards fingerprinting adversaries that are attempting to compromise a system.

Second, although there exists no prior work in the literature for fingerprinting SSH endpoints, a selection of previous studies have developed methods for detecting SSH brute forcers.

Hellemons et al. [7] have proposed an intrusion detection system method for detecting SSH intrusions using netflows. Similarly, Najafabadi et al. [14] propose a machine learning algorithm to detect brute force attacks in netflow data. The authors have validated their results on the SSH protocol and found that machine learning techniques perform well for detecting these brute force attacks. Nicomette et al. [15] clustered adversaries together based on attempted passwords. The authors find relationships between dictionaries, but at the same time notice that few dictionaries are shared across attackers. All these works focus on the detection of brute forcing attacks at the moment that the system is already under attack, however we show in the following that it is possible to obtain much information about the incoming request during the connection negotiation itself and before the password prompt is shown.

A selection of studies have investigated adversarial behavior after the successful compromise of a honeypot. Ramsbrock et al. [16] followed compromises made into four honeypots, and was able to derive a state machine to describe the actions of adversaries. Barron and Nikiforakis [2] investigated whether adversarial actions differed based on environmental factors, for example depending on the presence of real users on the systems and their usage of files. They were able to distinguish between human and bot login activity, and noticed humans did to a limited extent show interest in stored files while bots generally avoided significant interaction with the file system, and in half of the cases only proceeded to install a proxy gateway.

While proposed methods can identify brute force attacks, they do not allow for tool classification or for pre-emptively stopping these attacks. Given the current threat landscape, in which there is a high number of attackers, identifying attacks in an early stage before actual compromization is increasingly important. By forcing attackers to change their tools every attempt, the cost for attackers increases and many attackers will be priced out of the system. In this paper, we propose a method to fingerprint tools in use by adversaries, which can be used to track their activities over time, relate distributed attempts to the same toolchain and possibly actor, and thus gain a greater insight into the ecosystem as a whole.

3 The SSH Protocol

The secure shell protocol (SSH) is an established protocol for accessing services on a remote host, which is secured by an authentication procedure. In order for an attacker to enter login credentials it is first necessary to set up a secure protocol connection as specified in RFC4253 [20]. The main steps in setting up a secure communication channel between the attacker and its target are shown in figure 2 and go through three main phases.

First, after a TCP connection has been established, both parties exchange the version of the SSH protocol they are

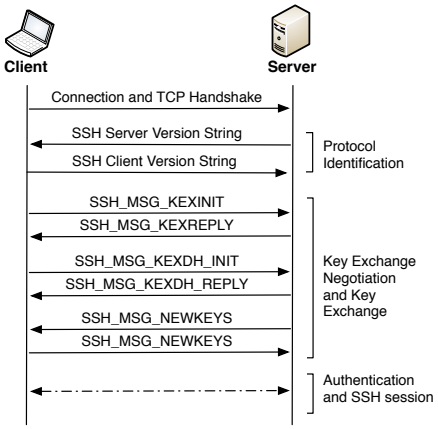

Figure 2: Schematic overview of the message exchanges in the establishment of an SSH session.

running in the protocol identification phase. Typical examples of sent version strings are `SSH-2.0-JSCH-0.1.51` and `SSH-2.0-paramiko_1.7.5`. As will later be explained, the version exchange is one of the components used for profiling attackers.

As SSH provides an authenticated encryption tunnel, both sides need to negotiate key material for the connection. In the second phase of the protocol, client and server negotiate the key exchange mechanism to be used. This negotiation is initiated by the client through a key exchange initialization message (SSH_MSG_KEXINIT), which contains all the different key exchange algorithms, encryption algorithms, algorithms to compute a message authentication code, and algorithms for compressing the data the client is able to accept. The order of the advertised algorithms is of importance as the algorithms are advertised according to the host's preference, and thus both the presence and order of algorithms shared during this step of the SSH connection can be used to profile a connecting client. After both parties have sent and received the key exchange initialization message, the highest commonly preferred algorithms are selected for setting up a secure connection. Depending on which algorithms have been chosen, the rest of the key negotiation and key exchange procedure slightly varies. After the key negotiation, the actual key exchange is initialized by sending the SSH_MSG_KEXDH_INIT message, after which the key exchange algorithm is run. Once the algorithm is finished, each side signals using a SSH_MSG_NEWKEYS message that the secure connection is set up and ready to be used.

In the third phase, both client and server switch to an encrypted tunnel using the just negotiated key material and perform the authentication, during which the client sends its login credentials. Given the correct credentials, the SSH protocol then makes the requested resource on the remote host available to the client.

4 Fingerprinting Tooling

Brute forcing the login credentials to gain access to a shell generally requires a great amount of attempts due to the large amount of possible username/password combinations. Therefore, it is uneconomical for an attacker to perform this task manually, and he or she will likely resort to a tool in order to automate the login attempts. Depending on the knowledge of the attacker, he or she will utilize an existing tool or develop a new one.

If the attacker opts to use known tools, there is no shortage of available material. A quick search on any search engine yields an extensive list of SSH brute forcing tools. Most of these programs are advertised as penetration testing tools, used to assess the security of a network, for example to find servers that use weak login credentials. More so, entire articles and tutorials are dedicated to the usage of those tools, such as Hydra [8], Medusa [13], and Ncrack [12].

When writing an SSH brute forcing tool, one could create an implementation of the SSH protocol, or use a pre-built library that handles the SSH connection. The aforementioned brute forcing tools utilize different libraries implementing the SSH protocol, and only add the logic to perform the attack. Although the libraries are likely to adhere to the standards specified in the RFC describing the SSH protocol, minor differences in connection establishment can be witnessed. One of those difference is the announcement of the SSH version. Libraries such as libssh [1] and libssh2 [17] use a different version string to announce compatibility with the same version of SSH protocol. Both libraries add their name and release version into the SSH version string; libssh version 0.7.1 announces `SSH-2.0-libssh-0.7.1`, whereas libssh2 version 1.8.0 identifies itself as `SSH-2.0-libssh2_1.8.0`. While this provides a first, trivial angle to identify the tools used by attackers, we only use this information as a reference and later combine it with the capabilities implemented by a specific library. Thus, if an adversary is spoofing the version string, the announced version will not match the fingerprint of the advertised key exchange, encryption, MAC and compression algorithms anymore, which in combination yields an even more distinctive fingerprint for a specific brute forcing tool implementation.

Similarly to the announced version string, the information exchanged during the key exchange initialization varies between libraries. As discussed in the previous section, during the session establishment, different algorithm suites are announced in order of preference. Not all libraries support all key exchange, encryption, MAC, or compression algorithms. More so, not all libraries supporting identical algorithms will order them according to the same preference. This multitude of possible variation of the initialization message increases the likelihood of libraries implementing them differently, which can be leveraged to identify the tool and/or underlying library in an incoming SSH connection request.

In order to compare different key exchange initialization messages, the four exchanged capabilities as discussed in Section 3 are used. The advertised

1. key exchange algorithms (kex),
2. symmetric encryption algorithms (enc),
3. message authentication code algorithms (mac), and
4. compression algorithms (comp)

are concatenated into a single capabilities string. Since we only care about exact matches in capabilities, we hash the concatenated string as a fingerprint for a specific tool. Consider the case of an out-of-the-box SSH server install on Ubuntu 16.04 Desktop, which comes preconfigured with the following configuration:

- *KEX algorithms:* curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-sha2-nistp521,diffie-hellman-group-exchange-sha256,diffie-hellman-group14-sha1
- *Ciphers:* chacha20-poly1305@openssh.com,aes128-ctr,aes192-ctr,aes256-ctr,aes128-gcm@openssh.com,aes256-gcm@openssh.com
- *MAC algorithms:* umac-64-etm@openssh.com,umac-128-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-sha2-512-etm@openssh.com,hmac-sha1-etm@openssh.com,umac-64@openssh.com,umac-128@openssh.com,hmac-sha2-256,hmac-sha2-512,hmac-sha1
- *Compression algorithms:* none,zlib@openssh.com

Thus, we obtain the fingerprint through the MD5 hash of

```
curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh
-sha2-nistp384,ecdh-sha2-nistp521,diffie-hellman-gro
up-exchange-sha256,diffie-hellman-group14-sha1;chach
a20-poly1305@openssh.com,aes128-ctr,aes192-ctr,aes25
6-ctr,aes128-gcm@openssh.com,aes256-gcm@openssh.com;
umac-64-etm@openssh.com,umac-128-etm@openssh.com,hma
c-sha2-256-etm@openssh.com,hmac-sha2-512-etm@openssh
.com,hmac-sha1-etm@openssh.com,umac-64@openssh.com,u
mac-128@openssh.com,hmac-sha2-256,hmac-sha2-512,hmac
-sha1;none,zlib@openssh.com
```

resulting in `9f735e5485614bcf6e88b9b848582965`.

Finding or building a tool for SSH brute forcing is only the first step for an attacker towards compromise - next the adversary needs to choose which login credentials to use during authentication. Here the attacker has three main choices:

First, an attacker can opt to incrementally, pseudo randomly, or randomly generate the username and password from a predefined set of characters. Given enough time, and assuming the attacker is not blocked after some attempts, this method is bound to provide access to the targeted machine. Second, an attacker can decide to generate the login credentials using a dictionary attack. In a dictionary attack, words are combined to form the password and or username. Third, an attacker can choose to use predefined login credentials and available password lists. As with SSH brute forcing tools, a quick search on any search engine reveals a plethora of hits advertising password list for download. These lists often contain known default passwords and usernames such as `admin` and `root`.

Here, again, the different options available for the attackers to generate the login credentials offers an opportunity to detect brute forcing campaigns, and we put forth the assumption that the same adversaries will most likely use the same or similar login credentials when trying to exploit systems. Under the premise that an attacker will use the same tool to attack multiple targets, the same fingerprints for a single IP address should be witnessed at different honeypots. While the combination of algorithms provides already a distinctive fingerprint for a specific tool, in the later part of our evaluation we further investigate the relationship between identical tools and password generation algorithms. If an attacker uses multiple machines or IP addresses with the same brute forcing tool and password generation algorithm, the previously described fingerprints can be used to cluster IP addresses belonging to the same attacker.

5 Data Collection

This section describes the setup of our honeypot infrastructure and the data acquisition strategy used in this paper. As the goal of our study is to demonstrate the possibility of fingerprinting the tools and techniques used by adversaries in SSH break-in attempts, we have designed a distributed honeypot system and exposed approximately 4,500 honeypots distributed over three /16 subnets to the open Internet.

5.1 Honeypot design

As you recall from section 3, the SSH protocol goes through three main phases in connection establishment: first, client and server announce their protocol versions to each other, second, the endpoints exchange their ciphering, MAC and compression capabilities and agree on a key, and finally, the tunnel is authenticated by a public key or password before an SSH session is established. While we would clearly expect to see differences in the behavior of adversaries after they have gained access to a particular machine, the SSH protocol is complex enough and contains several configuration options so that the tooling used by attacker may contain implementation differences, that will – as we show in the following – allow recognition of a particular tooling *even before* the SSH protocol advances to the password prompt and an interactive session. The distributed infrastructure was therefore implemented as a honeypot that would negotiate a key exchange and an SSH session with the connecting client, display a login prompt and collect usernames and passwords, but never let any user in. While this simplifies containment and thus reduces the risk of operating such a system, to the connecting user it basically appears like a regular server and an incorrect password guess.

In order to pose as yet another open SSH server, it is essential that the honeypot itself blends in with existing installations found on the Internet, otherwise knowledgeable adversaries could soon identify instances running some honeypot software and avoid individual IPs or even subnets where honeypots were detected. Thus, it would be a major failure if a system meant to identify adversaries based on handshake fingerprints could be fingerprinted itself, which has been found to be an issue for existing common open source honeypots such as Kippo or Cowrie [19]. To avoid this problem, we connect the incoming session to an actual OpenSSH implementation running inside a container, which matched in terms of version strings, list of available algorithms and options and ordering a default Ubuntu 16.04 LTS installation. This way, an adversary probing the system for implementation deviations to unexpected inputs will observe no difference from a typical server, and to an adversary scanning the Internet for banners and key exchanges our honeypots will blend in with what one would expect when connecting to a popular deployed operating system.

5.2 Organizational Placement

Aside from being identifiable based on specific implementation characteristics, it would also be conceivable that adversaries could spot honeypots based on the way they are deployed and subsequently avoid them. For example, an apparently open SCADA system hosted on an Amazon EC2 cloud IP address should trigger some suspicion in a knowledgeable adversary. In our case, an entire block of consecutive IP addresses running SSH servers where otherwise nothing else is open in the network could similarly bias the results, and adversaries be motivated to evade such networks.

In order to create a believable posture and collect representative results, we deployed the honeypot in the enterprise network of an organization. This organization is connected with three /16 networks to the Internet, on these networks approximately 60,000 devices are active and incoming SSH traffic is not filtered by the firewall. These 60,000 official network hosts were of various types and origin, with a mix of servers, workstations, laptops and other mobile devices. While the foremost category would be constantly powered on and accessible, personal workstations, laptops or phones would only be powered on at select times.

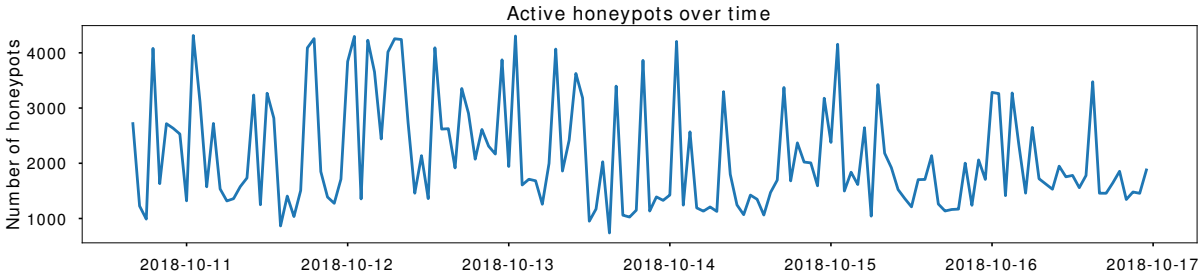

Figure 3: Number of active honeypots aggregated by hour over the course of one week in the study.

In this study, we spread the 4,500 honeypots randomly throughout the network ranges of the organization, so that they would be assigned IP addresses belonging to server, workstation or mobile host subnets. This meant that IP addresses belonging to our honeypot system were located in between sets of servers or regular user machines, thus an adversary exploring the network could not evade the honeypots by skipping select parts of the network ranges, and after scanning several "official" servers our honeypot would appear to the adversary as just another server in the same group. The routing rules of the organization were set up in such a way that IP addresses that were allocated to a host but *also* chosen for the honeypot were forwarded to the official host whenever powered on, and forwarded to the decoy as soon as the official host left the network. This way, adversaries interacting with the organization's hosts would experience an instantaneous seamless handover to our honeypot infrastructure.

Figure 3 depicts the number of active honeypots in our deployment over the course of one week. The graph clearly shows a diurnal pattern stemming from a base population of approximately 1,500 active systems in server and workstation ranges, as well as an additional 3,000 decoys which only become activated when the mobile and temporary devices leave the network. Both components of our honeypot deployment strategy make it thus very challenging for an adversary to locate and evade our infrastructure.

6 Evaluation

This section evaluates the results of applying the fingerprinting methods on the dataset. Both the SSH versions string and the fingerprint based on the session negotiation are analyzed. In addition, we use time and password correlation to evaluate the hypothesis that attackers leverage multiple hosts to brute force SSH servers.

6.1 Available fingerprints

During the data collection period, a total of 107,793 hosts tried to brute force the login credentials of one or more honeypots in our network. We only considered source IPs that completed at least one completed SSH key exchange towards our 4,500 honeypots during the entire month, thus excluding mere TCP SYN scans. Within this entire dataset, we observed a total of 123 SSH version strings, and identified 49 distinct MD5 hashes for different libraries and library versions in use.

While the analysis yielded a substantial count of different fingerprints and version strings, we also find that these instances are also surprisingly well spread across source hosts. The distribution of source IPs that use a particular fingerprint follows an exponential decay; while the most commonly used library fingerprint is used by 58% of the sources, already the bottom half of the top 10 fingerprints account only for fractions of a percent. Fingerprints thus have large amounts of variations, and are distinctively correlated to sources: more than 89% are only associated with one fingerprint over the entire observation period. Similar results apply to the version strings; the top 3 version strings are used by more than 75% of all source IPs, also here 90% of all sources only advertise one version string to our honeypot during the entire period.

The large body of fingerprints compared to the number of available tools, as well as the larger number of version strings compared to the amount of fingerprints matches our expectations how brute forcing tools are developed and used. First, as commonly used tools build on system libraries such as libssh or OpenSSH, major updates to the underlying system library implementing the SSH protocol will result in a new fingerprint, even though the adversary uses the same toolchain.

Second, tools (or their users) actively change the version string and configuration advertised by their toolchain, possibly in an attempt to evade detection by signatures. Examples of this are invalid encryption and mac algorithms such as `hmac-sha\x11` and `"lowfIsh` in some of the key exchange messages. Software packages such as OpenSSH allow a user to configure the ciphers used in the key exchange message, even if these algorithms are not implemented in the library itself. Inspection of the source code of for example the libssh2 library reveals that the versions strings announced by the majority of the hosts matches with the one in the source code. When we further analyze the design of available brute

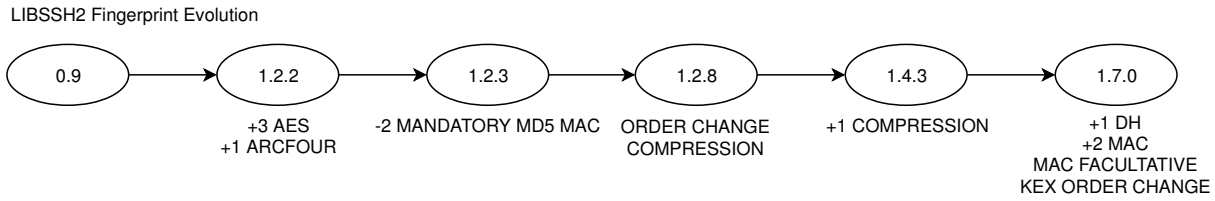

Figure 4: Evolution of the algorithms used by the libssh2 library to construct the SSH_MSG_KEXINIT message.

forcing tools, we can link common tools back to these libraries. Medusa [13], a tool for SSH brute forcing included in Kali linux, builds on this library in this selected version. The analysis of the 35 million compromisation attempts using the SSH version and key exchange fingerprinting leads to the conclusion that the majority of attackers use readily available tools to perform the brute forcing.

6.2 Fingerprints and libraries

Different tools generate different combinations in terms of SSH version strings and key exchange fingerprints. If two tools use different libraries implementing the SSH protocol, the announced SSH version string will likely be different. As seen in the data, the name of the SSH library is often included in the announced SSH version string.

Similarly, the fingerprint retrieved from the key exchange can also provide an indication of the tool used. Tools building upon different libraries will announce different algorithms during the key exchange. This is due to different libraries supporting different algorithms and announcing them in different order of preference. Therefore, due to slight variations in used libraries and implementations, tools can be linked to the fingerprints generated by processing the SSH version string and the key exchange initialization message.

In order to investigate the origin of the fingerprints and study user customization, we downloaded 10 commonly used SSH brute forcing tools to inspect which library is linked to realize the SSH protocol, and in addition mined the SSH version strings from all received handshakes for mentions of libraries or implementation stacks. This yielded a total of seven libraries that are utilized across SSH brute forcers we observed, namely (1) Granados, (2) JSCH, (3) libssh, (4) libssh2, (5) OpenSSH, (6) paramiko, and (7) the Erlang standard SSH implementation. For all these libraries, we downloaded every single release, as well as every intermediate version available in the software repositories and manually identified the location in the source code responsible for the SSH connection and parameter selection. A program then identified every intermediate/release version when this code was modified over the entire period of the softwares' past development, and we manually analyzed each changed code segment to extract how this library would advertise itself as in this particular version. This yielded a set of (banner, MD5 hash) tuples, which is on the one hand dependent on the version of a library, on the other hand also on certain options and taken branches in the code, for example if certain other libraries or headers were available on the system where it would be compiled and thus activate `ifdef` blocks. Based on this, we generated a set of possible 57 banner-hash tuples for the seven aforementioned libraries, implemented in a particular software package at any point in time.

Figure 4 shows this evolution for the libssh2 library with respect to the construction of the SSH_MSG_KEXINIT message. While libssh2 saw several intermediate releases between version 0.9 and version 1.2.2, and the library advertises itself differently in between these two releases, the cryptographic routines remained unchanged during all of these updates leading to an identical fingerprint. In 1.2.2, three options for the data encryption using the AES cipher suite as well as the option to encrypt using RC4 were added, while in 1.2.3 the previously mandatory option to create a message authentication code using MD5 was removed. Later versions such as 1.2.8 only differed in the order they advertised the preference of algorithms. Similar version graphs were created for the other six libraries mentioned above, most of them appeared in our data set announcing different release versions.

When libraries are compiled, the supported algorithms might differ, depending on installed software packages that are required by the algorithms. These dependencies can greatly affect the number of supported algorithms during the key exchange initialization, which is why we have identified all different combinations possible. These also result in a unique fingerprint for a particular installation path and thus allow a peek into the configuration of the attack hosts.

Given the advertised library and version string we can then cross validate whether the fingerprint obtained from the handshake is consistent with the default behavior of the library, or whether some code changes or configuration changes were introduced. Interestingly, when we look at the 123 SSH version strings and 49 fingerprint hashes that we collected in our honeypots, we find that all 57 theoretically possible tuples from the software libraries were present. When we match the version string for a particular library and the fingerprint hash that *should* have been generated from the honeypot handshake,

we find that there is only a match for 26 out of the checked 57 library versions. This indicates that in 31 instances, more than half, the announced version string is spoofed. We manually verified these instances of spoofing, and found that while some of them are attempts to make the version string more generic, others modify the version number of a library or pretend to run a different software stack than the behavior of the library would in practice indicate. For example, a particular brute forcing tool would announce OpenSSH version 4.3, but announces a cipher suite that was not implemented in this particular version. Also, the order of algorithms for some advertised versions of libssh and libssh2 do not match the implementation of the library.

While spoofing of version string is common among attackers, given our tracking of code changes, we were able to trace back 26 out of the 31 spoofing instances to a library that is consistent with the behavior of the brute forcing tool. Overall, we find that we were able to identify more than 91% of the tools used to attack our honeynet using the fingerprint.

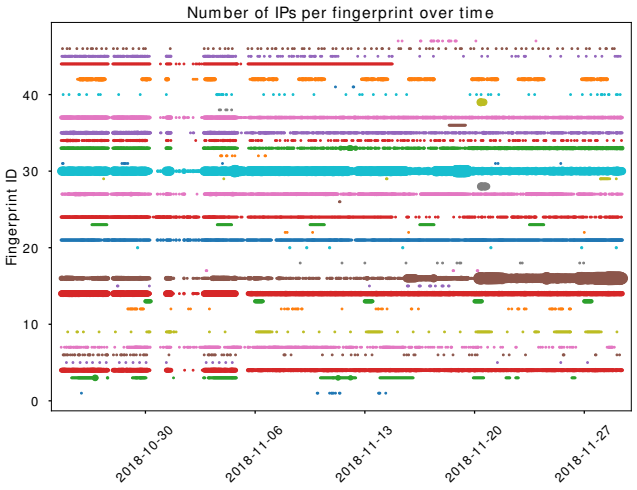

Figure 5: Scatter plot showing the number of hosts using a certain key exchange fingerprint over time. The number hosts is indicated by the size of the markers.

6.3 Collaborating hosts

As we have shown in the previous part, the combination of available key exchange algorithms, cipher suites, MAC and compression options together with the advertised version string does contain large amounts of entropy. As an additional verification that this fingerprint can serve as a measure to fingerprint the tooling itself, we look in this part into the behavior of the hosts exhibiting a particular fingerprint. If this relationship holds, we would expect the following two results: First, commonly available tools should see continuous usage, but within this set there could be groups of hosts that use the same tool in a specific way or with a similar behavior that could be clustered together. Second, given that we identified 31 mismatching version strings and fingerprints, we would expect some adversaries to have built custom tools for SSH brute forcing. As these are not publicly available, they should only be in used by a limited group of source hosts, thus the tool fingerprint could be used as a proxy to partially fingerprint the actor.

In the following, we will now investigate the different behaviors of the 49 different key exchange-cipher-MAC algorithm hashes that we initially discovered in our dataset. Figure 5 shows the activity of the these fingerprints over the course of the experiment, for compactness of the figure and the discussion, each fingerprint has been assigned a numeric ID from 0 through 48. The number of hosts using a tool with a specific fingerprint at a given time is represented by the size of the marker in the plot, with the area of the markers being proportional to the number of unique hosts using a fingerprint per hour. In the figure we can readily identify five distinct behavioral patterns of fingerprint usage:

- *Popular, commonly available tools* such as Ncrack (fingerprint 30 in cyan), or SSHtrix (fingerprint 16 in red) or Hydra (fingerprint 4 or 14 depending on the operating system it is installed on such as raspbian), see some continuous usage by a diverse and significant number of origins. In the next section, we further show that these groups can be separated by the password list configuration of the brute forcer into distinct subgroups that pursue a common strategy.

- *Custom tools* with a relatively uncommon or unique fingerprint are essentially only in use by distinct groups of adversaries. Often these are rolled out to a large amount of hosts, and from there explore remote hosts simultaneously with synchronized start and stop times. Consider for example fingerprint #23 indicated in green, which performs a weekly recurring scan of the address space, always using a similar amount of resources. Similar behaviors are shown by the groups of hosts using the tool with fingerprint #42 (orange) who scan every other day, or by the groups scanning bi-hourly using the tool with fingerprint #46 (brown).

- Among these, some *custom tools are run at low intensity* only by a select group of origins, which on average make less than 4 login attempts per hour. All installations with fingerprint #36 (purple) operate in this way, and may be classified as a slow brute forcing campaign following the description by Javed and Paxson [10].

- *Distributed hit & run campaigns* only occasionally surface, but then for a short period of time brute force many remote hosts with a large number of resources. Interestingly, we observe these hit & run campaigns to typically exhibit a unique fingerprint and thus employ custom

tooling for their activities, which makes these attempts and hosts participating in them easily identifiable by our proposed method. Examples of such fast, concentrated attempts are fingerprints 39 (light green) or fingerprint 29 (gray). The plot shows that both clusters become active during the same time period, around November 20th, and have approximately the same size. A closer inspection reveals that the hosts using the tool identified by fingerprint ID 29 are also using the tool identified by fingerprint ID 39. The IP addresses are located in 103 different /8 subnets indicating that the attacker has the knowledge, resource and intention to spread his or her infrastructure across the Internet, possibly in an attempt to avoid detection.

6.4 Password combinations

After the completion of the key exchange and session negotiation, the adversaries were presented a login prompt they could interact with. Past work such as Nicomette et al. [15] have used the entered user credentials to link individual login attempts into related clusters, and for example identified relationships between dictionaries but also noticed that only few of them were reused across adversaries. As discussed before, the economies of scale would imply that an attacker is most likely going to deploy the same setup and tooling on different hosts to launch attacks, thus the same tooling would result in an identical fingerprint, and thus help us gain deeper insights into the activities of the attackers, for example if they are splitting and distributing parts of dictionaries for brute forcing across collaborating hosts. In this section, we will discuss the relationship between groups found based on an identical banner and key exchange, and their associated password lists.

For each of the 49 fingerprints detected earlier, we extracted all SSH sessions from any host that exhibited this signature and assembled the set of credentials (username + password) during login attempts. Table 6.4 shows a selection of 8 fingerprints, which exemplarily shows the spectrum of different key exchange - password list behaviors found throughout the dataset. From the data we can distinguish three types of groups: First, we see clusters where tools and the credential list used are tightly linked together. Second, we clearly see select fingerprints in wide use, which focus on (subsets of) fixed password lists. Third, we observe groups of tooling, where hosts pursue brute forcing with diverse and customized password lists. We will discuss each of these three categories in the following subsections.

6.4.1 High credential / tool correlation

For each group advertising the same banner and using the same key exchange algorithm, the table lists the number of IP addresses matching this fingerprint and the number of unique login credentials list used by an attacker belonging to the cluster. To provide a better understanding of the login credential lists used, the number of hosts using the 5 most frequently used credential lists are shown.

All clusters in this category exhibit a tight link between the fingerprint and the utilized password list. For example, the fingerprint `0df0d56bb50c6b2426d8d40234bf1826` of cluster 1 is sent by 684 hosts, however within this group only 9 different password lists are used. The vast majority of hosts in this cluster, 672 or 98.2%, always send the exact set of credentials to our honeypots, deviations of the cluster default occur only very infrequently among all remote hosts having connected to our honeypots. In addition to the strong link from a particular fingerprint to a credential list, also the reverse is true: no other attackers have been using this credentials list in the dataset. This would indicate that the proposed fingerprinting method can be used as a predictor for password usage.

6.4.2 Popular tool

The second category of behaviors we can distinguish in the fingerprint analysis are those tools which are widely deployed but are run similarly configured. In this particular category, we observe the presence of multiple common credential lists, from which hosts pick a subset and brute force all of our honeypots with the same credential set.

An example of this is cluster 5 with 86,805 IP addresses, which employed a surprisingly low number of 625 login credentials lists. The top five groups all settle on a permutation of *(admin, admin), (admin, default), (admin, password)* and move horizontally throughout our ranges. Other groups of hosts choose from lists geared towards specific type of devices, for example credential lists associated with common IoT devices or Raspberry Pi distributions.

6.4.3 Diverse credential lists

While both previously described clusters could also have be discovered using password-based grouping as adversaries shared significant credentials, we found a third behavior of brute forcing which would have remained undetected to established methods. Clusters 6 through 8 belonged to a new category characterized by a high number of credentials list and a low number of IPs using identical credential lists.

Consider for example the case of group 6, where 557 different password lists appear across 564 different hosts. This is due to the fact that attackers in this cluster use random or pseudo randomly generated passwords in combination with 22 fixed credential tuples. The generated passwords are all 10 characters long consisting of letters and numbers. Next to randomization, each host only performs a limited amount of login credentials, 80% (463 out of 564 IPs) use less than 60 unique login credentials. Another indicator of randomization is the low overlap between passwords, 6,592 of the 10,318

Type	High credential / tool correlation			Popular tool		Diverse credential lists		
Banner	SSH-2.0-OpenSSH_7.4p1 Raspbian-10+deb9u3	SSH-2.0-OpenSSH_7.4p1 Raspbian-10+deb9u4	SSH-2.0-libssh2_1.8.1_DEV	SSH-2.0-libssh2_1.7.0	SSH-2.0-libssh2_1.8.0	SSH-2.0-Go	SSH-2.0-Go	SSH-2.0-libssh-0.6.3
Kex	0df0d56bb50c6b2426d8d40234bf1826	0df0d56bb50c6b2426d8d40234bf1826	1616c6d18e845e7a01168a44591f7a35	a7a87fbe86774c2e40cc4a7ea2ab1b3c	a7a87fbe86774c2e40cc4a7ea2ab1b3c	c39f4cec145ee3d50fb590595143b9d5	72d744cee7c48197c1b56973e8600140	51cba57125523ce4b9db67714a90bf6e
Cluster	6	7	8	4	5	1	2	3
IP count	684	1138	85	6473	86805	564	208	4479
# cred. lists	9	5	7	2688	635	557	111	4438
Credential list								
Top 1 list	672	1127	79	3602	64140	3	65	18
Top 2 list	2	6	1	16	6024	2	11	3
Top 3 list	2	3	1	16	4488	2	7	3
Top 4 list	2	1	1	13	4347	2	5	3
Top 5 list	2	1	1	10	2832	2	3	2

login credentials are used by only one IP address. The password randomization is confirmed by a mean Jaccard index of 0.4 of the credential lists used in the cluster.

While randomization of credentials would make it challenging for a password-based clustering algorithm to detect similarly acting groups, the same will hold true for brute forcers that rely on a *very* long lists of candidate credentials from which username/password combinations are picked. This will dilute possible linkage from the perspective of common credentials, however a clustering based on the fingerprint will find these relationships.

In this section we have shown that clustering based on SSH banners and key exchange algorithms can find different types of clusters. The proposed method is able to find groups using extensive password lists or random password, whereas password-based solutions would struggle. However, password-based solutions can provide better clustering performance for popular tools having multiple credentials lists such as cluster 5, hence a combination of both a fingerprint-based and password-based approach promises to provide more, and complementary findings.

7 Conclusion

In this work, we have described a method for fingerprinting tools for SSH brute forcing based on version strings and advertised algorithms. As this method distinguishes tools based on the data exchanged during the key initialization, we are able to detect tools prior to even entering the session authentication, and can deploy the mechanism transparently without any changes necessary to an existing enterprise infrastructure.

We have deployed the fingerprinting method over 4,500 honeypots for a period of one month, and from 35 million login attempts been able to detect 49 different fingerprints. The results indicate that different tools are used in different ways, indicating that attackers customize, or develop their own tools. Looking at the behavior of different tools, we were able to identify clear timing patterns originating from different tools, while based on timing patterns we can detect distributed brute forcing campaigns. By fingerprinting the tools used in a campaign, we are able to track and analyze the campaign over time. Additionally password analysis of detected clusters allowed us to identify different brute forcing methods. Both assessments contributed in providing insights into the tactics, techniques and procedures of the attackers.

Future Work

The work presented in this paper was conducted from the angle of cyber threat intelligence, with the aim of augmenting the portfolio of methods to fingerprint adversarial tooling and gather insights into their activities and behavior. This study led to a variety of different fingerprints, some of which could be traced back to specific brute-forcing tools. In principle, the contribution of this method could however be wider, and potentially be suitable as an additional detection rule within the context of intrusion detection systems. To evaluate its merit for such active threat detection and prevention, further research is however needed to evaluate its efficacy, which has not been done within the scope of this study.

References

[1] Aris Adamantiadis, Andreas Schneider, Nick Zitzmann, Norbert Kiesel, and Jean-Philippe Garcia Ballester. libssh. `https://www.libssh.org/`.

[2] Timothy Barron and Nick Nikiforakis. Picky attackers: Quantifying the role of system properties on intruder behavior. In *Annual Computer Security Applications Conference*, 2017.

[3] David J. Bianco. The pyramid of pain, 2013.

[4] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. The security impact of https interception. In *The Network and Distributed System Security Symposium*, 2017.

[5] Vincent Ghiette, Norbert Blenn, and Christian Doerr. Remote identification of port scan toolchains. In *IFIP International Conference on New Technologies, Mobility and Security*, 2016.

[6] Vincent Ghiette and Christian Doerr. How media reports trigger copycats: An analysis of the brewing of the largest packet storm to date. In *ACM SIGCOMM Workshop on Traffic Measurements for Cybersecurity (WTMC)*, 2018.

[7] Laurens Hellemons, Luuk Hendriks, Rick Hofstede, Anna Sperotto, Ramin Sadre, and Aiko Pras. Sshcure: a flow-based ssh intrusion detection system. In *Conference on Autonomous Infrastructure, Management and Security*, 2012.

[8] Marc Heuse, David Maciejak, and Jan Dlabal. Hydra. `https://github.com/vanhauser-thc/thc-hydra`.

[9] Martin Husák, Milan Čermák, Tomáš Jirsík, and Pavel Čeleda. Https traffic analysis and client identification using passive ssl/tls fingerprinting. *The European Association for Signal Processing Journal on Information Security*, 2016.

[10] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. In *ACM Special Interest Group on Security, Audit and Control Conference on Computer & Communications Security*, 2013.

[11] Maciej Korczyński and Andrzej Duda. Markov chain fingerprinting to classify encrypted traffic. In *IEEE International Conference on Computer Communications*, 2014.

[12] Gordon Lyon and Fotios Chantzis. Ncrack. `https://nmap.org/ncrack/`.

[13] Joe Mondloch. Medusa. `http://foofus.net/goons/jmk/medusa/medusa.html`.

[14] Maryam M Najafabadi, Taghi M Khoshgoftaar, Clifford Kemp, Naeem Seliya, and Richard Zuech. Machine learning for detecting brute force attacks at the network level. In *International Conference on Bioinformatics and Bioengineering*, 2014.

[15] Vincent Nicomette, Mohamed Kaâniche, Eric Alata, and Matthieu Herrb. Set-up and deployment of a high-interaction honeypot: experiment and lessons learned. *Journal in Computer Virology*, 2011.

[16] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling attacker behavior following ssh compromises. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.

[17] Daniel Stenberg, Marc Hörsken, Viktor Szakats, and Will Cosgrove. libssh2. `https://www.libssh2.org/`.

[18] Qixiang Sun, Daniel R Simon, Yi-Min Wang, Wilf Russell, Venkata N Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy*, 2002.

[19] Alexander Vetterl and Richard Clayton. Bitter harvest: Systematically fingerprinting low- and medium-interaction honeypots at internet scale. In *USENIX Workshop on Offensive Technologies*, 2018.

[20] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol. Technical report, Internet Engineering Task Force, 2006.