SSH Honeypot: Building, Deploying and Analysis

Harry Doubleday, Leandros Maglaras*, Senior Member IEEE, Helge Janicke
School of Computer Science and Informatics
De Montfort University, Leicester, UK

Abstract

This article is set to discuss the various techniques that can be used while developing a honeypot, of any form, while considering the advantages and disadvantages of these very different methods. The foremost aims are to cover the principles of the Secure Shell (SSH), how it can be useful and more importantly, how attackers can gain access to a system by using it. The article involved the development of multiple low interaction honeypots. The low interaction honeypots that have been developed make use of the highly documented *libssh* and even editing the source code of an already available SSH daemon. Finally the aim is to combine the results with the vastly distributed Kippo honeypot, in order to be able to compare and contrast the results along with usability and necessity of particular features. Providing a clean and simple description for less knowledgeable users to be able to create and deploy a honeypot of production quality, adding security advantages to their network instantaneously.

*Principal corresponding author
Email address: leandros.maglaras@dmu.ac.uk (Leandros A. Maglaras),

Introduction

There has been a variety of honeypots previously developed to work using the SSH protocol. The aim of this article is not to build software that can better these is every way, but more of a focus on a quick, simple, yet effective alternative to the pre-built packages available as well as providing a piece of software that can be available to professionals and unenlightened server users en masse.

A honeypot is a wittingly vulnerable piece of software or system that is often used to emulate a service, system or network. The advantages of honeypots are that they are intentionally exposed in particular ways. The ruse and falsification used in honeypots is to hopefully entice attackers, which can be harder than it may seem as most attackers with some sort of knowledge, not a 'script kiddie', will soon realise that they are not in a real system when they try to run certain commands or processes that the honeypot doesn't understand. The results from different types of honeypots often vary significantly in depth, which will be further discussed in the results section of this document. (Spitzner (2002)) states that, a honeypot should be available to be attacked, as a security resource it has no value or purpose when it is not probed, attacked or compromised. The results that are produced from honeypots can cause vast improvements in computer security, including but not limited to; improved Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS) and Anti-Virus software. However, arguably the most important feature is that, when emulating a particular service or system, the honeypot is configured exactly the same as the regular services running on the system. The reason for this is that if an attacker succeeds at breaking into the honeypot with the same configuration it is very likely that the actual service could be compromised and is in need of some extra protection (Jonsson et al (2004)).

There are two main categories of honeypot that this article is concerned with and they are often used to gather very different information about the attacker. Low interaction honeypots, which can be referred to as *facades,* are much simpler to build and maintain, as they tend to be a simulation of a particular service, such as SSH (HKSAR (2008)). Low interaction honeypots have been favoured by the industry due to the simplicity and ease to set up and collect meaningful results (Provos et al (2008)). The limitations involved with these particular honeypots are vast as they only emulate a specific service and often will have no system beyond that particular service. Although they have their limitations, these types of honeypots have been the most prolific in recent years due to these limitations. The reason for this is that the user of this type of honeypot will be able to collect and analyse data that is only relevant to the service they are concerned with, which can give a much

deeper understanding of the techniques and patterns that attackers tend to follow.

High interaction honeypots are what most people would consider as a typical honeypot. They provide a fully functioning system that will allow the attacker to interact with the system on all levels. Quite simply a high interaction honeypot can be any vulnerable system that is connected to a network and can be monitored for analysis. (Semantic (2008)) describe these as truly vulnerable systems that can be probed, attacked and exploited, once the attacker gains access to the system the honeypot can be used in a botnet or to carry out other attacks. This gives light to some ethical issues with regard to continuing the research once a honeypot has been compromised, when should the system be taken back from the attacker and should it really be used in the type of attacks that it has been designed to prevent? It is for this reason that they take a lot of maintaining and will also need a system such as Honeywall, a gateway service monitoring all traffic, in order to complete a full forensic investigation. The example used throughout this article has been Kippo, which was deployed for this project. Other than interaction levels, honeypots can be classified in other ways such as; usage, virtual or physical.

Honeypots can take many forms and this means that they are regularly deployed in very different circumstances and positions within networks. They must also take into account the complexity of what they are researching, for example certain pieces of malware will not act in a malicious way when it finds itself in a virtual environment, this is obviously because the more we are allowed to research the methods that attackers use the more they must evolve in order to maintain the allusive nature and evade detection (Zakorzhevsky (2010)). One of these methods is the minefield deployment system; this method will have a honeypot which is placed within the same subnet as a number of servers giving a better chance that the attacker will alert the honeypot if trying to breach a server on that system. It is well known that most attacks will scan an entire network or range of addresses and honeypots within this range will notice this scan, even if they use tools slowing down the scans to try and prevent the IDS from being alerted (Jonsson et al (2004)). Other mechanisms of deployment include a Honeynet (https://www.honeynet.org/about), this is a method of deploying an entire network of honeypots, which individually can collect information about particular services and as a whole can provide details on what is most likely to be attacked and whether the attacker will attempt to sit in the network attempting to perform attacks such as Man in the Middle.

Amazon Web Service

The hosting of this article was done on the Amazon Web Service(AWS). AWS provides a number of services but the Elastic Compute Cloud (EC2) is the web service which was used. EC2 provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers and it is very useful for deploying honeypots. A main benefit of the AWS is that with its elastic computing it allows the volumes of instances to be attached, detached and reattached to instances. Being able to detach and reattach a volume may seem unnecessary but should the user become locked out of an instance, because of configuration modification, the whole server is not lost. One of the main issues surrounding honeypots is that if they are not attacked they are of no use (Spitzner (2003)). The AWS, being part of one of the largest companies in the world, has a very high amount of traffic through its web servers and attackers know the range of IP addresses, making it much more likely that they honeypot will be able to collect an adequate amount of data. The AWS allows the user to select a particular region for where their cloud servers are deployed, putting it in a different bracket of IP addresses, which could give massively different results. The SmartHoney article has used AWS for running all manner of honeypots, focused on various services, one in particular is SSH where they found that placing their honeypots in certain regions meant a significant variation in the volume of these attacks (https://blog.smarthoneypot.com/tag/aws/). Considering the use of AWS has been very beneficial to much larger and full time honeypot projects; SmartHoney, Secure Honey it seems that it should more than suffice for a much smaller similar project.

SSH Protocol

The SSH protocol is designed to give the user a secure method of connecting to a system, to login or use the other services on a system, over an insecure network. (Ylonen et al (2006)). The SSH protocol uses a three step process in order to create the secure session; these steps are as follows, SSH transport layer, SSH user authentication and SSH connect. These steps are in fact sub-protocols that run on top of the previous sub-protocol respectively to create the SSH tunnel.

The transport layer is the first sub-protocol when creating an SSH session, using TCP/IP to connect to port 22 of the server in order to provide authentication of the server and the key exchange. After the initial connect message there is a protocol-identification so that both parties are using the same protocol, SSH version 2 for example. The key exchange algorithm is then negotiated between the client and server and then the key exchange itself takes place using the agreed algorithm. (Poll et al (2011)).

The user authentication process is the server confirming the identity of the user attempting to gain access. This can involve various methods, but must always include the public key authentication. (Ylonen et al (2006)). This is a check between the server and the client that the respective public and private keys are owned as this is used to encrypt the messages. Public key encryption uses two mathematically related keys, public and private, in order to encrypt and decrypt data. The private key is secret and only the owner should know it, whereas the public key is made readily available. Anything encrypted using the public key can only be decrypted using the corresponding private key and visa versa. (Comodo. (2015)). Although this is the most secure method of authentication it is not always enabled and can sometimes be bypassed if the server will accept password authentication instead.

The final sub-protocol is the SSH connect, which runs on top of SSH transport layer and SSH user authentication. This sub-protocol is used to create channels used for data transfer, where each terminal session, forwarded connection, etc, are separate channels that are multiplexed into a single connection. It can provide channels for login sessions, TCP/IP connections and allows remote command execution along with file transfer using SFTP. (Ylonen et al (2006)). SFTP is not to be confused with FTPS, many things have changed since the introduction of protocols such as FTP and sending data over any public network without a form of encryption is considered very dangerous and in some cases prohibited. Regulations like PCI-DSS and HIPAA, for example, contain provisions that require data transmissions to be protected by encryption. When regulations such as these were initially discussed it was obvious for the need of a secure way to transfer files, which gave light to the Secure Socket Layer (SSL) being used on top of FTP to create FTPS. The issue with this is that it requires a minimum of two channels, one for the initial connection and subsequent commands and one for and data transfer, which causes a higher risk of a security breach as there must be a range of open ports on each system. SSL also does not offer any authentication per se as any certificates used can be self signed, therefore this is not an efficient method to determine the authenticity of any persons or servers that are being communicated with. Whereas SFTP uses only one channel as previously discussed to tunnel all information through. SSH is more specifically for remote login and has almost completely replaced Telnet for command-line access to remote computers.

Brute Force Attacks

The most common form of initial attack involving SSH is brute force and in fact it is the most prolicfic form of attack against Internet facing servers, (Owens 2008; The SANS Institute 2007). The concept of a brute force attempt is simple; try every possible value until authentication has been achieved. The issue with using brute force is that given a 5 character password, where only letters that are of the same case are used, it could take 26^5 guesses (11,881,376). Given that the majority of passwords contain more letters and/or use numbers or special characters, the amount of time taken to gain entry could easily surpass the attackers lifespan. In order to speed up this process and make it worthwhile for an attacker they will often use large lists of common passwords, called dictionaries. Dictionary attacks can be significantly much more efficient than brute force attacks because they are not sequentially trying password combinations but rather, known common passwords that are widely used. By default most SSH servers will have a limit to the number of authentication attempts that can be tried per connection, but as with many things involving connectivity it can be bypassed by the attacker, if the correct configuration is not it place, quite simply by adding an extra parameter to the initial connection command:

```
ssh -lusername -oKbdInteractiveDevices=`perl -e 'print "pam," 
x 10000'` targethost
```

The above command would allow the attacker up to 10000 password attempts before the connection is refused, which obviously is very useful while undergoing a brute force attack. (http://arstechnica.co.uk/security).

Building and Deploying

The aforementioned low interaction honeypots developed have been written in the C language, this is because there is a large amount of documentation involving available libraries, such as libssh, functions and source code that are readily available for inspiration and utilisation.

There are many different ways to go about creating a low interaction honeypot of production standard, but with the aim of being simple to use and develop while maintaining the effectiveness of result gathering it can be a difficult trade off. The first method that was used was similar to many projects that already exist using the C SSH library, libssh, to employ the fuctions of the SSH protocol.

While conducting initial research about the SSH protocol and involved honeypot projects, there were quite a few production honeypots that are

available and as most of these are open source projects the source code can be easily attained and edited to improve or configure on the users specific system. The most notable of these actually used the libssh for C was the SecureHoney project, which had modified a honeypot that has been previously written by another developer. This type of method to produce a honeypot is useful and most of all safe for the user to run, the reason for this is that the connection is never actually authenticated. The program uses the functions in the libssh library in order to listen for connections and begin the authentication process. The information gathered about the attacker is written into a file for later analysis. Issues with this is that an attacker with the know-how will realise that this is not an SSH daemon because information regarding the SSH can be collected while scanning and interrogating before attempting an attack. Given this information it was evident that, while this was exactly the type of honeypot that was to be produced during this project, an alternative to this could provide arguably better results with substantially less programming and development.

The alternative idea however does not emulate the SSH daemon, because it was created by editing the source code of by far the most prolific SSH daemon in use, OpenSSH. OpenSSH was originally part of the OpenBSD suite. Considering that in 2008 OpenSSH had 88% of the market share and in October 2015 announced that it will be natively supported on windows (Microsoft, (2015)). The advantages of this are that the honeypot will be, to all intents and purposes, an actual version of the OpenSSH daemon. This means that an attacker is much less likely to be susceptible to suspicion when attempting to brute force the system.

Although this seems like a honeypot in the loosest of senses, it can be very beneficial as a production honeypot. As the software can be configured to provide an output very similar to that in the SecureHoney project, including creating specific files for logging attempts and even collecting IP addresses of the attackers. There are many problems that can occur when attempting to use this method, as the source code for the daemon is being edited and recompiled, including making it difficult to actually use the SSH service for anything other than they honeypot, which can be devastating if this is being performed to a remote server.

Methods

Honeypot in C

The first step to this process was becoming familiar with the libssh and the functions that were imperative to creating a valid SSH session that we would need as a basis for the honeypot. These functions are an example of how the libssh functions can be used to set up the standard configuration of a new

SSH session, which include;

```
static ssh_session session;
static ssh_bind sshbind;
session=ssh_new();
ssh_options_set(session, ssh_options_timeout, &timeout)
sshbind=ssh_bind_new();
ssh_bind_options_set(sshbind, ssh_bind_options_banner, "ssh \r\n");
ssh_bind_options_set(sshbind, ssh_bind_options_bindaddr, listenaddress);
ssh_bind_options_set(sshbind, ssh_bind_options_bindport, &port);
ssh_bind_options_set(sshbind, ssh_bind_options_hostkey, "ssh-rsa");
ssh_bind_options_set(sshbind, ssh_bind_options_rsakey,rsa_keyfile);
```

The next step after making sure that the session has been set up and is listening on the desired port we must be able to accept incoming connections and drop them after the user authentication credentials that the attacker used have been logged and placed in a file called ssh_attemps. This forms the basis of the honeypot and used sections of an SSH honeypot that was found at as it fulfills the task of collecting the password attempts.

Modification of OpenSSH

This section is to describe exactly how the daemon can be modified to create a honeypot that is easy to maintain with little coding, although this can all be bypassed entirely by simply running the script that has been developed to automate the process. The automation of this via a script makes this method more efficient than developing a honeypot in C, having said this, the OpenSSH source files are written in C and manual editing of this would need some level of knowledge regarding programming in C.

This method has been separated into two separate methods, this is because there is an instance where both methods could be doubled together in order to gather much more information from a selection of servers.

The first way of doing this is to simply modify the source code of the daemon. By doing this no SSH connection attempt will be authorised, the attackers IP address along with the username and password that was attempted, and these connection attempts will be written to a file, in the /var/log directory, called ssh_attempts.

```
                File *logFile;
                logFile = fopen("/var/log/sshd_attempts", "a+");
                fprintf (logFile, "From: %s at: %s | user: %s, pass: %s
\n", get_remote_ipaddr(), authctxt->user, password);
                fclose (logFile);
```

The most important part of this code is the *return 0;* segment, which is within the password authentication file in the source code. This line means that no matter what is entered by the attacker the authentication will always result in a failure. The problem with this method is that by doing this, the sshd is rendered useless for any sessions that the user may need in future without reverting the modifications.

The second method when editing source code requires a little more setting up and involves a second instance of the SSH daemon. The reason for this is that having the service running twice as two separate services allows different configuration for each daemon, therefore one should be configured as the façade daemon and one should be configure as a usable service. The usable service should be placed on a large port number preferably between 10000 and 65535 and designed with usual SSH security.

Finally, using a combination of both daemon modification methods a network of servers could each run multiple SSH daemons. Unlike the previous method though, this method has two fully functional daemons, one of which can be used by the user for their normal SSH activity and the other uses the *ForceCommand* in the sshd_config file. This will force all connections that are attempted on this daemon, to a central server that is running the aforementioned modified daemon that accepts no connections and logs all attempts, including IP address, username and password.

Analysis

While running various honeypots, that have been partially developed or modified for the purposes of this project, the medium-interaction production honeypot Kippo was also deployed. The reason for initial deployment of this particular honeypot was to give a better understanding of the way that well known products, that are already available, record certain log attempts as well as the particular features that are available. This gives an insight into this type of technology available and provides an example of the reporting technique that's used. Another reason that this honeypot was deployed was to see if all the functions that are available in Kippo are of any use.

Interestingly the results of running this honeypot showed that large number of the attackers, once inside the honeypot, typed a single command and then exited. The command used wasn't always the same but the response given by the system obviously prompted the attacker to leave. From this given information, it was deduced that the attackers knew they were within a honeypot. After experiencing this a little more research was conducted, via the SANS institute forums, and it would appear that this behavior could be a

number of things, but most likely that they had in fact realised the honeypot for what it is. Accessing the server that is running Kippo can show this, and running the command that plays out a particular connection live.
$: ~/kippo/utils/playlog.py 20160316-100915-9940.log

```
finance:~# curl -o /tmp/u251 http://222.186.21.201:2014/u251
bash: curl: command not found
finance:~# chmod 777 /tmp/u251
finance:~# /tmp/u251
bash: /tmp/u251: command not found
finance:~# ubuntu@ip-172-31-10-173:~/kippo/log/tty$
```

Another idea is that this is part of an automated brute force attack. When the target system has finally been compromised, the machine that is conducting the attack saves the last password guess and logs out so that the owner can browse the compromised machine at their convenience. Another notable point was the large amount of IP addresses that had attacked this honeypot were predominantly Chinese and South Korean based internet service providers. This was also the case with downloads, using *wget* command. The downloads were directed to servers with Chinese IP addresses, many of which had been blacklisted online by various sites that provide lists of malicious hosts and reports it to relevant bodies. (https://cymon.io/222.186.15.61)

Kippo is a good tool but observation proves that fingerprinting may mean that by using a medium interaction honeypot such as this, we may not actually gain any better results than the low interaction SSH honeypots that never accept connections. Kippo can be difficult to use properly as a server admin with little experience of this type of technology, with more dependancies and longer set up time along with much more maintainence for sql databases, whereas a method that doesn't bother with what an attacker might possibly do once inside and a purely keeping them out strategy could provide just as valuble information with ease.

The idea was to use this as an inspiration in order to create something similar but more refined to my particular needs. It is for this reason that I used a similar recording approach in other methods for collecting the results. Although this honeypot has been successful with previous projects, it seemed to give a fair few problems when I attempted to run this on one of my AWS instances. Naturally there were some dependencies to install and some configuration of the honeypot that was necessary before it could be used. The issues faced with running this on an AWS instance were initially compatibility errors. Errors including being unable to install a fully functional version of OpenSSL, which is a dependency with all SSH services as the libraries are used, this was resolved by using a different AWS instance because the package could not be located and installing from source on the server did not compile. More problems followed this, once the honeypot could be compiled

and built it still wouldn't run due to the program being unable to find the private key file. This took up a vast amount of time to resolve as all the configuration and code, to my knowledge, was correct. Although I had a lot of problems with this technique, I believe that from a development point of view it is by far the most powerful because it gives the developer free reign to create a honeypot to their choosing and functional requirements. On a final negative point, this technique should be used to create much more powerful projects and programs such as Kippo, when attempting to use such sophisticated techniques to emulate a daemon it makes no sense to limit the service by not implementing it into a medium to high interaction honeypot.

When emulating a service is required it seems to be far more efficient to modify a daemon that already has an enormous market share. Modification, as can be seen in this project, can be just as useful as developing a honeypot from nothing, if not more so because of the time saving. The reason that the method of two SSH daemons was used is because it allows the most amount of modification if necessary, as it is the source code being modified. This also makes the honeypot instance of the daemon incredibly secure, as the password authentication will always fail regardless of what is entered by the attacker. However, this procedure also offered some difficulties, such as modifying the incorrect files or missing out very necessary steps in the process. The issues prompted a fair amount of time studying the man pages for various configuration files, along with researching the innumerable Linux forums for clarification on certain files, including their contents and location. Fortunately all attempts to create this method were conducted within a virtual machine, which prevented the implementer from being locked out of my AWS instances when restarting the SSH daemons. A solid understanding of the protocol, daemons, libraries and system files is necessary for developing any of these previously discussed honeypot designs.

Conclusions

Although this article has seemingly concluded with a tool that offers very similar services to those that are already available, this is by no means the limit to what is possible. Further work would involve the creation of a bash script. This script could then be used by 3rd parties who wish to conduct this sort of research or as an easier option when waiting to launch an SSH honeypot. Other possible development opportunities could include making this honeypot more available as a production honeypot. As the software that has been modified is open source, the redistribution of modified versions of it is permitted under its license (St. Laurent, (2008)). Therefore it would not be

difficult to produce a script that automates the whole process, using *wget* to obtain the modified code. The benefit of this easy method of install means that it could easily be placed on a large group of servers. Speculatively speaking, this would give light to even further development, using the sshpot.com as stimulus. The group of servers that are running the modified daemons would send the results to a main hub of results, being able to produce statistics and security enhancements alike. A thought on how this would be achieved, would be running a chronjob that ran another script. This script would check the hash of the sshd_attempts file and forward the results if any new ones had been recorded. Alternatively, editing the sshd_config file once again could also do this. These new additions would include a Match Group User section added that forced all connections made, to the modified daemon, straight to the main server utilising the ForceCommand option. Rather than beginning with a complete new build that is a honeypot, use existing well developed and highly distributed tools in order to develop a instrument that could be used on a commercial scale

References

Chamotra S, Bhatia JS, Kamal R, Ramani AK (2011), Deployment of a low interaction honeypot in an organizational private network. In: International Conference on Emerging Trends in Networks and Computer Communications, pp 130–5.

Chowdhary V, Tongaonkar A, Chiueh T (2004) Towards automatic learning of valid services for honeypots. In: Proceedings of the 1st International Conference on Distributed Computing and Internet Technology, New York, p 469.

Cymon, malicious IP address database, available at: https://cymon.io/222.186.15.61

Grimes RA (2005) An Introduction to Honeypots, Apress, Berkeley

The Government of the Hong Kong Special Administrative Region(HKSAR), Honeypot security, 2008. Available at: http://www.infosec.gov.hk/english/technical/files/honeypots.pdf

HoneyNet project, https://www.honeynet.org/about

King Cope, OpenSSH bug Available at: https://kingcope.wordpress.com/2015/07/16/openssh-keyboard-interactive-authentication-brute-force-vulnerability-maxauthtries-bypass/
http://arstechnica.co.uk/security/2015/07/bug-in-widely-used-openssh-opens-servers-to-password-cracking/

Microsoft, 2015, available at: https://blogs.msdn.microsoft.com/powershell/2015/10/19/openssh-for-windows-update/

Mokube I, Adams M (2007) Honeypots: concepts, approaches, and challenges. In: Proceedings of the 45th Annual Southeast Regional Conference, New York,pp 321–62.

Nicomette V, Kaaniche M, Alata E, Herrb M (2011)
Set-up and deployment of a high-interaction honeypot: experiment and lessons learned.

Owens, J., A Study of Passwords and Methods Used in Brute-Force SSH Attacks, 2008

Poll, E and Alesky Schubert, Rigorous specifications of the SSH Transport Layer, Technical Report ICIS-R11004, Radboud University Nijmegen, 2011

Provos N, Holz T (2008) Virtual Honeypots: From Botnet Tracking to Intrusion Detection, Addison-Wesley,Boston.

Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004. edited by Erland Jonsson, Alfonso Valdes, Magnus Almgren

SANS Institute. 2007. SANS Top-20 2007 Security Risks (2007 Annual Update). Available at: http://www.sans.org/ top20/ 2007/

SecureHoney project, available at: http://securehoney.net/, https://github.com/sb542/SecureHoney, https://github.com/PeteMo/sshpot

SmartHoney project, available at: https://blog.smarthoneypot.com/tag/aws/

Spitzner L (2003) History and Definition of Honeypots, Pearson Education, Boston.

SSH Assigned Numbers (RFC 4250)

SSH Protocol Architecture (RFC 4251)

SSH Authentication Protocol (RFC 4252)

SSH Transport Layer Protocol (RFC 4253)

SSH Connection Protocol (RFC 4254)

St. Laurent, Andrew M. (2008). Understanding Open Source and Free Software Licensing. O'Reilly Media. p. 4. ISBN 9780596553951.

Zakaria WZA, Mat Kiah ML (2012) A review on artificial intelligence techniques for developing intelligent honeypot. In: Proceedings of the 3rd International Conference on Next Generation Information Technology, Seoul, pp 696–701.

Zakaria WZA, Mat Kiah ML (2013) A review of dynamic and intelligent honeypots doi:10.2306/scienceasia1513-1874.2013.39S.001, ScienceAsia, 39S