The paper begins on the next page.
Before you begin reading, we have a request...

Please Help Make This Study Better. Take Part by Contributing Data!

The data in our study is collected by a script that has been carefully designed to anonymize data so as to protect your privacy while providing us with only enough information to publish aggregated data on the potential threat described in our paper. By contributing to this study, you help to ensure that our results best reflect SSH usage in environments like yours. Further, we encourage you to run the script on as many hosts as possible as this will greatly improve the quality of our data set and accuracy of our results. Anyone on a Macintosh (OS X or higher), UNIX, or UNIX derivative (BSD/Linux/Sun, etc.) operating system can participate.

To get the data collection script, download
http://nms.csail.mit.edu/projects/ssh/collect-ssh.tar.gz
If you have wget, you can do this from the command line using the following command:

```
wget http://nms.csail.mit.edu/projects/ssh/collect-ssh.tar.gz
```

Then, execute the following four commands to configure the script.

```
gzip -d collect-ssh.tar.gz
tar xvf collect-ssh.tar
cd collect-ssh
sh build-CR.sh
```

Finally, run the script with the following command. If you are an authorized administrator of this host, please consider running the script as root so that we can collect the full set of data from all `ssh` user configurations on the host.

```
perl collect-ssh.pl
```

You will then be shown the data being collected. When collection is complete, you will be asked if you are willing to submit it to us and prompted for a transmission method. If you are behind a firewall, we recommend email submission. Regardless of how the data is transmitted, it will be encrypted first. If you experience trouble in your environment with the default options, read section II of the README file that comes with the script for more information.

Further information about protecting your site from information harvesting by patching an existing OpenSSH installation or upgrading to OpenSSH 4.0 is available on our project website at
http://nms.lcs.mit.edu/projects/ssh/

We greatly appreciate your support!

Inoculating SSH Against Address-Harvesting Worms[*]

Stuart E. Schechter ses@ll.mit.edu
Jaeyeon Jung jyjung@mit.edu
Will Stockwell bigwill@mit.edu
Cynthia McLain cdmclain@ll.mit.edu

Abstract

Address harvesting is the act of searching a compromised host for addresses of other hosts to attack. Secure Shell (SSH), the tool of choice for administering and communicating with mission-critical hosts, security-critical hosts, and even some routers, leaves each user's list of previously contacted hosts open to harvest by anyone who compromises the user's account. Attackers have combined address harvesting with myriad mechanisms for impersonating a host's legitimate users to obtain a remote shell via SSH. They have succeeded in breaching systems at major academic, commercial, and government institutions. In this paper, we detail the threat posed should attackers automate this mode of attack to create a self-propagating worm. We then present a countermeasure to defend against address harvesting attacks, with an implementation written for OpenSSH. We also present the first study to measure how much information is available to attackers who harvest addresses from users' `known_hosts` databases and search for unencrypted identity key files. We found that a surprisingly large fraction of the `known_hosts` entries were to hosts on distant networks, that the bulk of these entries could be reached by compromising a small fraction of the user accounts in our survey, and that 62.8% of identity keys encountered were stored unencrypted.

1 Introduction

The SSH protocol has done much to popularize the use of cryptography for remote command execution, file transfer, and other services. However, cryptographic channels alone are not enough to ensure these services will only be accessed by their intended users for the purposes they authorize. As SSH has become one of our most trusted services, attacks that highlight its limitations have become widespread. A small set of attackers have exploited weaknesses in SSH authentication practices to impersonate legitimate users and compromise systems at a large number of major universities, corporations, national laboratories, supercomputing centers, and even military installations [8, 15, 29, 19].

One reason these attackers have been able to target such a large number of institutions is that SSH clients store `known_hosts` databases, which map the list of remote hosts each user has previously contacted via SSH to their public keys. When a host is compromised and an attacker learns how to impersonate one of its users, the list of addresses in the `known_hosts` database is easily harvested for use in targeting other hosts. Such reliable target lists reduce both the time required to find vulnerable hosts and the likelihood that attacks will raise alarms due to failed connections or authentications. Security practitioners have observed attackers using the `known_hosts` database to identify target hosts for future compromise [8].

In Section 2, we will describe a number of weaknesses in authentication and credential management practices that expose SSH servers to impersonation attacks, such as the re-use of weak passwords on multiple hosts and reliance on the operating system to protect identity key credentials. Impersonation mechanisms that exploit these weaknesses include attacks that use stolen authentication credentials and that insert fraudulent credentials into password files or `authorized_keys` files [19]. We describe how these impersonation attacks could be used to construct a worm in Section 3.

In Section 4, we describe how these machanisms were used to carry out the recent attacks. We also present recent trends in malicious code that indicate these same techniques may soon be fully automated to create a *worm*, or self-propagating malicious program. These trends include the use of automated tools to perform online dictionary attacks against SSH and the emergence of worms that perform online dictionary attacks on other protocols.

To better understand the consequences of attacks that

[*]This work is sponsored by the United States Air Force under Air Force Conract F19628-00-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

harvest addresses from SSH, we have initiated the first multi-institution study, on SSH `known_hosts` relationships and key management, collecting data from 2,077 user accounts on 92 hosts. We use the data from this study, presented in Section 5, to explain how these `known_hosts` databases have enabled attackers to repeatedly compromise host after host, and network after network.

In Section 6, we discuss the countermeasures that can be used to safeguard against address harvesting, as well as the trade-offs they require. As a result of this work, one of these countermeasures has now been implemented into OpenSSH 4.0.

2 SSH Impersonation Attacks

Before we can adequately describe the recent attacks on SSH and the potential of SSH worms, we must first explain the mechanisms that can be used to impersonate users. These mechanisms require no protocol or software vulnerability in SSH. Instead, an attacker who compromises one user account on a host can employ other exploits to compromise other user accounts on that host. The mechanisms described below, and summarized in Table 1, leverage access to one compromised user account on a source host in order to enable the attacker to impersonate that user when authenticating to a target host on which that user also has an account.

2.1 Exploiting misplaced trust

SSH servers and user accounts are often configured to trust other hosts to act on their behalf, to authenticate users, or to safely store user credentials. All of these practices are potential targets of attack.

T1 — Exploiting reliance on other host's security
If an attack comes from a compromised host that is listed in the `shosts.equiv` or `hosts.equiv` file in the target server's `/etc` directory, or the `.shosts` or `.rhosts` file of the targeted user, the attacker will be permitted to connect to a target user's account without presenting user credentials.

Even if no hosts are explicitly trusted to authenticate on behalf of the target host, such trust is often implicit. Many users place their public identity keys in their `authorized_keys` files on SSH servers and leave their secret identity key unencrypted on hosts they use as SSH clients, trusting that these accounts will not be compromised. If one such client account or host is compromised, then the attacker can read the unencrypted identity key and use it to authenticate to the target host.

T2 — Abuse of forwarded authentication agent
Authentication agents are programs employed by users to authenticate on their behalf. They free users from the need to retype the pass phrases that protect their identity-key credentials each time that they authenticate.

A user can configure his agent to authenticate on his behalf when accessing services from an application run on a remote host. However, most SSH agents do not verify that the actions a remote host performs are the actions the user intended to authorize. Thus, when the user believes he is authorizing a CVS transaction he may instead be authorizing an SSH connection to a host targeted by the attacker.[1]

2.2 Credential theft

An attacker who can obtain a user's credentials can impersonate that user on any host that accepts these credentials. An attacker may choose from any of a number of approaches to steal credentials.

C1 — Password theft by compromised SSH server
When authenticating via passwords, the SSH client will send the user's password credentials to the server over an encrypted channel. When the user's password arrives, it is then decrypted into plaintext before it is checked against the password file. If the server belongs to or has been compromised by the attacker, then the attacker can modify the SSH server to collect these passwords. The attacker can then proceed to gain access to other hosts on which this password is used for authentication.

This attack can be thwarted if the client is configured to authenticate via a challenge-response protocol, such as SSH identity-key authentication or the Secure Remote Password (SRP) extension [26].

C2 — Extraction of keys from authentication agents
To free users from the need to retype the pass phrases that protect their identity key credentials, an authentication agent must keep these credentials in its memory.

Once an account is compromised, an attacker can search the process table for active authentication agent processes. He can then copy, dump, or directly inspect the

[1]The agent in Michael Kaminsky's remote shell client, REX [11, 12], provides a partial solution to this problem by verifying that the service being authorized (but not the command or parameters passed to the service) is indeed the one that the user intended.

memory space of those processes to which he has access in order to locate identity key credentials.

C3 — *Online* dictionary attacks
An online dictionary attack is staged by repeatedly attempting to authenticate to a remote host using common passwords. Intrusion detection systems can be trained to detect these attacks and terminate communications with attacking hosts. However, if an attacking host is permitted to continue these attacks and chooses a large set of targets, it will eventually find servers that allow continued connection attempts and employ common passwords.

C4 — *Offline* dictionary attacks
After obtaining the password file on a compromised host, an attacker can test candidate passwords against the password file or try to decrypt identity key files in user home directories. While it is likely that an attacker who could access the password file could compromise this account without the password, chances are that the user employs this password to authenticate to other hosts as well. Such *offline* dictionary attacks also differ from their *online* counterparts in that the attacker need not run the authentication protocol. This is advantageous because executing a network protocol increases the risk that alarms will be activated and introduces a network delay for each password tested. Once a user's credentials have been compromised, the attacker can use them to gain access to other hosts on which they are accepted.[2]

C5 — Eavesdropping by client software or host
A patient attacker who has compromised a user's account can modify or observe the SSH client and agent to collect passwords and identity key pass phrases as the user types them. It can then either store the host address, username, and password triplets that it observes, or it can send them directly to the attacker.

Many users find it convenient or necessary to open SSH clients on hosts to which they are already connected via SSH. We use the term *gateway hosts* to describe those hosts to which a user connects via SSH from a client and from which the user then initiates a new SSH connection to another host running the SSH server. It is often necessary to use gateway hosts when firewalls prevent direct access from the user's immediate client to his or her desired destination host. SSH may also be employed to protect file transfers, CVS commands, or other services required by software that is run at a gateway host. Attackers can strike users on these gateway hosts even if an SSH server is not run on the user's immediate client.

[2]A 1995 study by Bishop and Klein [2] showed that 40% of passwords were crackable. More recent reliable statistics on the percent of crackable passwords are harder to find. Suffice it to say that while user awareness of weak passwords may have improved since then, the sophistication of cracking algorithms has also improved and the speed of computers used to crack passwords has followed the expontential growth of Moore's law.

2.3 Insertion attacks

An attacker may be able to insert his own commands into a user session or insert his own credentials in place of a legitimate user's credentials. The former attack, in which the attacker impersonates the user for part of the SSH session, can be used to perform the latter attack, which allows the attacker to impersonate the user in future sessions.

I1 — Session capture and command insertion
While proper use of identity keys, authentication agents, and agent forwarding can protect against credential theft at gateway hosts, these practices offer very little real protection if connection is routed through a compromised client or host. All communications are decrypted and then re-encrypted at the client, and software at this host can insert, modify, or delete information at will.

I2 — Credential insertion or replacement
An attacker can insert an identity key into the user's `authorized_keys` file. The SSH server depends on this file to determine which keys the user has authorized to serve as his credentials. If the compromised user's home directory is located on a shared file system, the attacker then uses the inserted identity keys to authenticate as that user to other hosts that mount the user's shared home directory.

If the attacker can write to the system password file, he can replace any or all user passwords with those of his choosing.

3 Components of an SSH Worm

The success of a worm depends on the number of hosts that are vulnerable to attack, the speed at which the worm can propagate, and its ability to evade detection to avoid triggering a response. Depending on the availability of certain classes of vulnerabilities and the skill of the author, a worm could target SSH using a variety of the attacks from Section 2, as summarized in Table 1.

To spread quickly, an SSH worm will need to infect as many new hosts as possible immediately after each host is compromised. Upon compromising a new host, such a worm could impersonate that host's users by tak-

	Attack	Event triggering attack opportunity	root not required	non-interactive	stealthy
T1	Unencrypted identity key file located	User's account or host compromised	*	X	X
T2	Forwarded agent used to authenticate attacker	Compromise of account or host already running forwarded agents	*	X	X
C1	Password stolen by compromised Web server	New password-authenticated session initiated to compromised server	X		X
C2	Identity key extracted from SSH agent processes	Compromise of host running agent processes	*	X	X
C3	Online dictionary attack on password file	Authentication protocol executed with correct username/password guess	X	X	
C4	Offline dictionary attack on passwords and identity keys	Password hash computation completed with correct password guess		X	X
C5	Password or key entered into previously compromised SSH client or agent	SSH client/agent executed on compromised host	X		X
I1	Session insertion attack	User's account or host compromised	*	X	X

Table 1: Attacks on SSH and the properties that affect their effectiveness when used in a worm. An 'X' indicates either that an attack can be run from a user account (*root not required*), need not wait for interactive user events in order to spread (*non-interactive*) or would not require excessive network traffic (labeled *stealthy*). A star (*) indicates that the attack can run without root privileges, but only against accounts available to the compromised user.

ing immediate advantage of any unencrypted identity keys (T1), extracting identity keys from running agents (C2), taking over any existing SSH client sessions (T2), and using forwarded agents to authenticate on its behalf (T2). Obtaining root access to the compromised hosts would enable these attacks to be carried out using data from all of the host's users, and would then allow the worm to begin an offline dictionary attack to obtain any credentials that it does not already have.

After the worm has exhausted all immediate targets, it could steal passwords from users that login to its SSH server (C1) and observe clients and agents to collect credentials (C5). While this may be unlikely to speed the overall spread, the worm can take advantage of any activity that may have caused it to be detected – the administrators may be next to login and his credentials may be the most valuable of all. While *online* dictionary attacks (C3) are likely the slowest and most overt, worms may still benefit from employing them after all other vectors have been exhausted.

Of course, a remote exploit in SSH that allows the attacker to impersonate any user could enable a worm to spread to the set of all accessible hosts running vulnerable SSH servers. Such a worm could spread unencumbered by the delays incurred by attacks that wait for user interaction, search for credentials, or repeatedly run authentication protocols. The speed and stealth of such a worm would be bounded by its ability to correctly identify and contact other vulnerable hosts.

Regardless of how a worm performs impersonation, it will need to identify target hosts. For attacks that employ stolen credentials or forwarded agents, it will need to identify hosts on which a specific user has an account. Attackers could not hope for a better repository of prospective target hosts addresses than that provided by the SSH client's `known_hosts` database.[3] For each user, this database stores addresses of the hosts to which the user has connected, each of which is mapped to the host's public key. Most implementations store this list sorted in the order in which the hosts were first contacted, allowing the attacker to first focus on those hosts that are newer and less likely to have been moved or retired.

The user's `known_hosts` database is not the only source of addresses of potential targets. When present, the administrator-configured global `known_hosts` file provides a list of targets that are likely to be receptive to connections from *any* user on the infected host. Once this wealth of information is exhausted, it may be possible to find more host names in configuration files and SSH server logs. These log files list, in plaintext, the names of clients that have connected to the server and the user accounts to which they connected.

In the upcoming sections, we will describe how attackers have used impersonation attacks to compromise remote hosts and used `known_hosts` databases to identify ad-

[3]While called the `known_hosts` database, OpenSSH and other implementations store this data in a flat file within a subdirectory of the user's home directory.

ditional targets.

4 Is an SSH Worm Imminent?

As automated patching has helped to reduce the availability of hosts with vulnerable software, attacks that target authentication mechanisms have been on the rise. As more attackers target SSH and these attacks become more automated, the onset of worms that attack SSH appears imminent.

Worms such as Lovgate [23], Deloader [21, 6], and Gaobot [22] already use *online* dictionary attacks in order to spread, though using protocols other than SSH. While such brute force attacks are among the least effective, they are frequently found in the wild because they are among the easiest to write.

While we are not aware of a worm that employs *online* dictionary attacks against SSH, *online* dictionary attacks targeting the protocol have been automated and made publicly available [18]. Evidence of their use appears in reports from the SANS Internet Storm Center [3] and from the anecdotal reports of security professionals and network researchers.

Trojaned SSH clients have also become widespread, and a number of these have been lifted from compromised hosts [4]. While such clients have been known to exist for some time, the incidents of attacks using them has been rising dramatically. In 2004, a number of high profile attacks were staged using trojaned SSH clients and *offline* dictionary attacks. Hosts were compromised at a large number sites including major universities, national laboratories, and supercomputing centers [15, 29], as well as major corporations and military installations [8]. Logs that recorded attackers in action show SSH connections to hosts in the `known_hosts` database being initiated immediately after the `known_hosts` database was read [8].

As a result of the attacks, some installations had to be taken offline for multiple days [15, 8, 29]. One of the educational institutions that contributed to our anonymized study reported that they had been forced to initiate a policy of disallowing all SSH connections from outside networks.

The scope of these attacks appears to be limited only by the time available to the attackers, a factor that would not constrain a worm. Given that most of the components a worm writer would require are already available, there may be little time left to improve our defenses before they put the pieces together.

In fact, such a worm attack would not be without historical precedent. The Morris worm of 1988 used *offline* dictionary attacks to crack passwords. The Morris worm also harvested target addresses from files such as `.rhosts` and `.forward` [20]. Because the Morris worm preceded the advent of SSH, the `known_hosts` database was not available to it.

5 Empirical Data

To better understand how an SSH worm might spread, we have undertaken a multi-institution effort to collect data from users' `known_hosts` database entries and their overall SSH configuration. We made available a data collection and reporting script, written in Perl, that could be run on each host either by individual users to collect data from their own account or by system administrators to collect data from all user accounts. The data collection and reporting script is publicly available at `http://nms.lcs.mit.edu/projects/ssh/`.

5.1 Collection methodology

A summary of the information submitted by our data collection and reporting script, `collect-ssh.pl`, is shown in Table 2.

All IP addresses collected, marked with a star (*), have been anonymized twice using the prefix preserving algorithm of Xu *et al.* [27]. The prefix preserving property ensures that two addresses within the same network before anonymization will fall into the network after anonymization. The addresses were first anonymized by the data collection script as it executed on the submitting host. The second anonymization step, performed by us after the data were collected, is necessary to prevent the anonymization key in that data collection script from being used to reverse the anonymization function and identify hosts in our published results. Public keys and usernames, marked with two stars (**), are replaced by their SHA1 [14] hashes.

When our data collection script runs on a submitting host, it queries that host's IP address and includes the anonymized address as part of the submission report. When we receive the submitted report over a TCP connection, we compare this submitted address with the IP source address field as seen by our servers. Collecting the address at its source allows us to differentiate hosts

host information	
	OS and version
	SSH and version
	IP address*
	Netmask
user identification	
	Username**
	IP address* of host exporting user's home dir
`known_hosts` file for each user	
	IP address* of each host for which key is
`authorized_keys` file for each user	
	Public identity keys**
identity key files for each user	
	Public identity keys**
	Flag: set if matching private key is encrypted
	SSH key version

Table 2: The contents of the report generated by `collect-ssh.pl`, organized by data source.

within a local network even if these hosts are behind a network address translation (NAT) box. Though the addresses are anonymized, we can still use the IP source addresses as seen by our servers to differentiate hosts on two distinct networks, even if they use the same local IP address behind their respective NATs.

Before submitting the report, the script encrypts the contents using a public key to ensure that the report cannot be read by an eavesdropper or an attacker who might attempt to compromise the server we use to store submitted data. Users can opt to send the encrypted report to our collecting server via either HTTP or SMTP, or they can save it to a file for manual submission.

5.2 Results

At the time of writing we have collected data from 2,077 user accounts that contain `known_hosts` files that were submitted from 92 distinct hosts. These files contain a total of 31,446 `known_hosts` entries to 8,009 unique destination addresses. Of those, 14 submissions came from hosts on which the collection script was run as root and on which data were submitted from all users. For 78 hosts, we received a total of 82 individual user submissions with at most two user submissions per host.

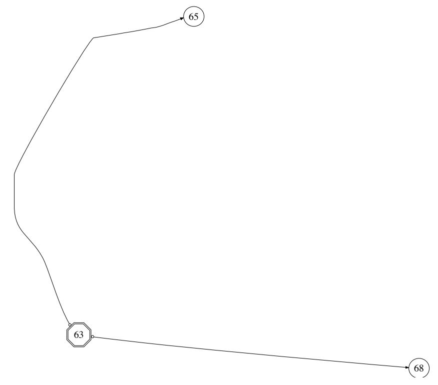
The median number of unique `known_hosts` addresses was 251 on hosts for which we collected data from all users, but only 24 for hosts on which we had to rely on individual submissions. Thus it is reasonable to assume that our data exclude a significant subset of the `known_hosts` entries on hosts from which we received individual submissions.

To illustrate the relationships between hosts represented by `known_hosts` entries, we generated graphs in which the nodes represent hosts from which we received submitted data. Each edge represents an entry, in a `known_hosts` database located on the host represented by the source node, that contains the address and key of the host represented by the destination node.

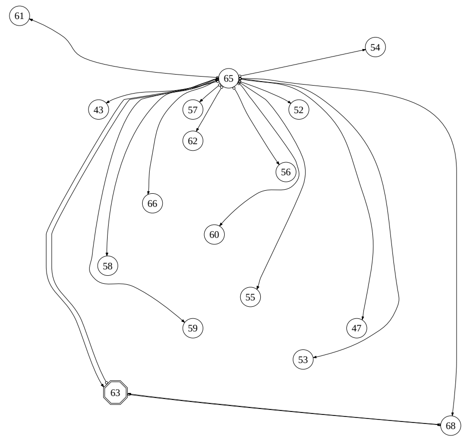
Figure 5 (attached as an appendix) is a graph of the `known_hosts` relationships within the institution from which we collected the most data. Of all the hosts in the graph, only the 3 hosts represented by rectangular nodes ran the script as root and provided us with their full set of `known_hosts` relationships. Even though we collected data from a subset users, themselves on a small subset of the hosts on the network, the connectivity of `known_hosts` relationships is quite extensive, spanning 1,290 nodes within the organization.

The set of nodes and edges visible in Figure 5 is also deceiving, as space constraints prevented us from displaying terminal nodes — those that are the destinations of `known_hosts` edges but from which we have not collected any data. We have placed below the label of each node the number of outgoing edges that would exist if terminal nodes were included in the local network graph.

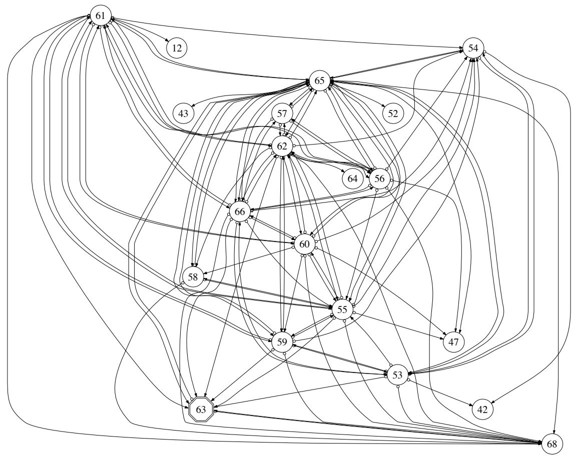
The series of graphs in Figures 1(a), 1(b), and 1(c), show the spread of `known_hosts` edges starting at a single host, node 63 in Figure 5, and spreading through the institution. Figure 1(a) shows the nodes at the institution that are destinations of the origin node's `known_hosts` entries. A total of six terminal destination nodes are not displayed. Figure 1(b) overlays the nonterminal nodes at the institution that are destinations of `known_hosts` entries of the hosts in Figure 1(a). Figure 1(c) shows the next iteration of these `known_hosts` relationships.



(a)



(b)



(c)

Figure 1: Host nodes that are reachable in 1 (a), 2 (b), and 3 (c) steps from a source host by traversing `known_hosts` edges.

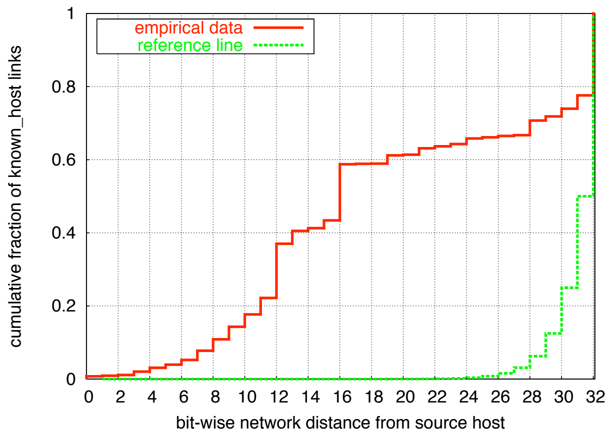


Figure 2: The fraction of `known_hosts` entries for which the destination host falls within a given bit-wise network distance of the source host. For reference, the dotted line represents, for a given source host, the probability that a destination selected uniformly from the IP space would fall within the given bit-wise distance from a source host.

Among the eight hosts that are a step away from our origin is one that itself has 574 outward edges. A total of 574 nodes can be reached with two steps from the origin, and an additional 177 can be reached with a third step. These graphs illustrate that, regardless where an attack starts, once a node with a large number of outward edges can be reached the attack can spread to the rest of the internal network in a very small number of steps.

To understand how extensively `known_hosts` relationships span organizations, we look at the bit-wise network distance between each `known_hosts` entry's source and destination host. For IPv4 addresses, bit-wise network distance is calculated by subtracting the number of common high order bits (prefix) from the length of the address (32). Figure 2 is a cumulative distribution function that shows the fraction of `known_hosts` entries that fall within a given bit-wise network distance. As expected, the distribution shows that `known_hosts` relationships are biased towards nearby hosts. Despite this bias, we observe that nearly 40% of the relationships span different /16 (Class B) networks and 30% span different /8 (Class A) networks, which helps to explain why SSH attacks can so easily spread between organizations.

In fact, the `known_hosts` relationships from this small set of hosts span 88 /8 (class A) prefixes, or 55% of all valid /8 networks.[4] Compromise of a single gateway

[4]The 160 valid class A networks are those that exclude two private /8 networks and 94 unallocated /8 networks, as documented by the

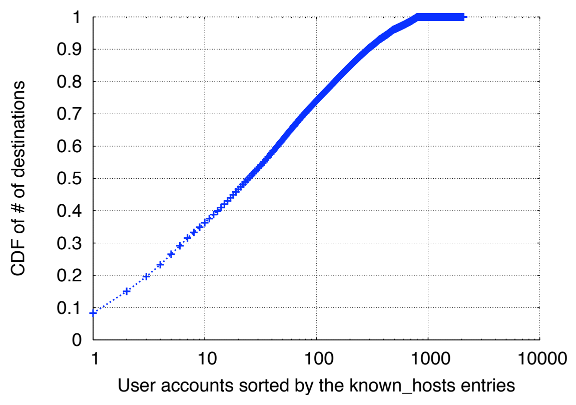


Figure 3: At any point $x = i$, the y value represents the fraction of destination hosts reachable from user accounts $1 \ldots i$, where the user accounts on the X axis are sorted such that the first account has the `known_hosts` file with the most unique destination hosts.

host can yield accounts on many networks. We identified a single host, outside of the network shown in Figure 5, that had `known_hosts` relationships with 74 unique /8 networks.

Figure 6 shows `known_hosts` relationships that span organizations. The nodes represent either distinct educational institutions (labeled .edu), firms (.com), or /8 networks (labeled with the most significant byte of the anonymized IP address). Once again, space constraints prevent us from showing terminal nodes.

Given the extensive interconnections between hosts, we were next led to ask just how much of this information a worm would need in order to spread. We observed a wide variance in the number of destinations in users' `known_hosts` databases. About 70% of user accounts have less than 10 unique destinations recorded in their `known_hosts` database, whereas over 10% of user accounts have used SSH to contact more than 50 distinct hosts. To assess the contribution of each user account to the set of unique `known_hosts` destinations, we sort the user accounts starting with the user account that has the most `known_hosts` entries. The second is the one with the most `known_hosts` entries to destinations not reached by the first user account, the third is the one with the most `known_hosts` entries not reached by the first or second user account, and so on.

Figure 3 shows that the number of distinct destinations grows rapidly, with the first 100 user accounts (less than 5% of the total) contributing 5,885 unique destinations (more than 74% of all unique destinations). This implies that it may be possible to compromise the bulk of all hosts by compromising only a small subset of user accounts.

Internet Assigned Numbers Authority [9, 10]

In addition to collecting `known_hosts` data, our collection script also checked to see if SSH2 identity keys were present and whether users had encrypted them with pass phrases. Of 274 identity key files collected, 172 (62.8%) were unencrypted and open to abuse by anyone able to read them.

6 Countering Address Harvesting Attacks

One way to thwart the spread of address-harvesting SSH worms is to hinder their ability to harvest target host names and addresses from the hosts they infect. If worms can be forced to resort to IP scanning to find targets, then they can be detected using existing scan detection techniques [16, 25].

These host names and addresses cannot be completely removed as they represent important information about the software's configuration before execution, state between executions (`known_hosts`), and history for forensic purposes (log files). Each type of file has different restrictions regarding which parties need to be able to read or write to it, as illustrated in Table 3, and so different solutions (or variants on solutions) are best for each. The most challenging file to manage is `known_hosts`, as it must be read and modified both by SSH and by those that use and administer it. We will take advantage of the fact that the common case is for the file to be read by SSH, and that users only need to access the file manually when locating, copying, or removing an entry.

	SSH		**User/Admin**	
	Reads	**Writes**	**Reads**	**Writes**
`known_hosts`	X	X	X	X
config files	X		X	X
log files		X	X	

Table 3: Of the files read or written by SSH that contain host names/addresses of other hosts, only `known_hosts` must be readable and writable by both SSH and its users.

6.1 Protecting `known_hosts`

To understand how SSH implementations could hide addresses in `known_hosts` databases, it is instruc-

	Contents of known_hosts entry	Harvest resistance	Additional usability cost
(0)	name, ip_addr, key	None	None
(1)	$(s_1, h(s_1 \circ \text{name}))$, $(s_2, h(s_2 \circ \text{ip\_addr}))$, key	Resists plaintext harvesting	New commands required to find/delete entries
(2)	$(s_1, h(s_1 \circ \text{name}))$, $(s_2, h(s_2 \circ \text{name}, \text{ip\_addr}))$, key	Resists offline dictionary attacks on IPv4 addresses	User can no longer locate entries in known_hosts using only their IP address
(3)	$(s_1, h(s_1 \circ \text{name} \circ \text{key}))$, $(s_2, h(s_2 \circ \text{name} \circ \text{ip\_addr} \circ \text{key}))$, date_and_time_entry_added	Resists offline dictionary attacks on the IPv4 address space and on host names	User can't distinguish between changed key from known host and new key presented by unknown host. Adds need for key revocation lists.

Table 4: A summary of possible organizations for SSH known_hosts entries, where h is a one-way collision-resistant hash function and s_1 and s_2 are randomly generated values (salts). Each approach is incrementally more resistant to harvesting than the one above it, but incurs an incremental cost in usability.

tive to look at how password databases evolved to defend against similar threats. Early multi-user computers stored passwords in plaintext files and, like known_hosts files, relied upon the file system to prevent their misuse by keeping them secret. In 1974, Evans, Kantrowitz and Weiss [5] proposed that passwords be hashed with a one-way function before being stored in the password file.[5] Their key observation was that the host did not need to store the passwords themselves, but only enough information to later verify that a password provided to the host was the same one the user had previously provided. Surely similar approaches can be used to protect SSH known_hosts databases.

We present three possible approaches with which one-way collision-resistant hash functions can be used to hide the identity of hosts in known_hosts databases. In Table 4 we summarize these approaches and contrast them with the original known_hosts format. Each solution is more harvest-resistant than the last, but this added resistance comes at a cost in usability.

Approach (1) – Simple name/address hashing

The simplest approach to prevent harvesting of plaintext host names and addresses is to hash their values as one would hash a password in the password file. The use of randomly generated salts, s_1, and s_2 ensure that the work required to stage a dictionary attack against one entry cannot be re-used on other entries. This simple hashing strategy can be summarized by the information stored in each known_hosts entry.

$$\begin{aligned}&(s_1, h(s_1 \circ \text{name}))\,,\\ &(s_2, h(s_2 \circ \text{ip\_addr}))\,,\\ &key\end{aligned}$$

We first implemented this approach into OpenSSH 3.9 using SHA1 [14] as our hash function h and base64 encodings of random 64 bit numbers as salts. In response to earlier drafts of this paper, the OpenSSH development team coded their own implementation of this approach, which first appeared in OpenSSH 4.0.

When the SSH client is called upon to initiate a new connection, it checks the destination host name and address against the known_hosts database entry by entry. A special string ('$<$' in our implementation and '$|1|$' in the OpenSSH 4.0 implementation) indicates that the host name or address has been replaced with a hashed token. In this case, the destination host name or address is hashed using the salt extracted from the token, base64 encoded, and then compared to the hash encoded in the token. Matching encodings imply with reasonably high probability that the addresses match. To maintain backwards compatibility with earlier SSH implementations, a plaintext comparison between addresses takes place when the address in the known_hosts database is not hashed.

Since entries in the known_hosts database are created and verified automatically by the SSH client, its behavior will remain unchanged from the user's perspective. We implemented two new commands for manipulating the known_hosts file should the user need to do so. remove-knownhost deletes a host entry from known_hosts by name and ssh-showkey returns the key of a host specified by name or address. In the

[5]For details on the adoption of this approach, see the early work of Robert Morris (Sr.) and Ken Thompson [13] or more recently Garfinkel *et al.* [7].

OpenSSH 4.0 implementation, these commands are integrated as options in `ssh-keygen`.

To speed the transition to hashed host addresses we provide a program, `ssh-hostname-encoder`, that hashes all of the addresses in an existing `known_hosts` file. In OpenSSH 4.0, this functionality is accessible via a command option in `ssh-keygen`. We have also provided a Perl script, `convert_known_hosts.pl`, that can be run to convert all `known_hosts` files on a given filesystem into hashed host address format. As no such script was provided by the OpenSSH 4.0 team for their implementation, we have provided one. It can be downloaded from
`http://nms.lcs.mit.edu/projects/ssh`

Approach (2) – Resisting IPv4 dictionary attacks

As with password files, the hashing approach is potentially vulnerable to an *offline* dictionary attack. On IPv4 networks, the attacker can expect to identify an IP address with a worst-case average of 2^{31} SHA1 calculations. While this might be time consuming enough to slow spread and raise alarms, an attacker can decrease the expected work by starting with addresses near that of the compromised host (recall Figure 2). All of the nodes on the victim host's class C can be identified by performing less than 256 SHA1 calculations for each `known_hosts` entry.

The possibility of dictionary attacks leads us to suggest that SSH client implementations may not want to store IP addresses at all. It should only be necessary to associate the key with the address used by the user on the command line, which is most often the domain name. If hashed IP addresses must be stored, than we propose that it should be salted both with a random salt and with the host name. This will significantly increase the cost of attack in networks where reverse DNS lookups are disabled, and increase the likelihood of detection where these lookups are enabled but monitored.

$$\begin{aligned}&(s_1, h(s_1 \circ \text{name}))\,,\\&(s_2, h(s_2 \circ \text{name}, \text{ip\_addr}))\,,\\&key\end{aligned}$$

Approach (3) – Resisting all offline dictionary attacks

Host names are also subject to dictionary attack, especially if common names such as "gateway", "mail", and "database" are used. A design approach to eliminate *offline* dictionary attacks requires more fundamental changes to way that SSH clients confirm that the host being contacted is indeed one that was last contacted at the same address. We propose that rather than storing entries that consist of hashed names and address mapped to the host's key, the SSH client should instead concatenate the host key onto the value to be hashed for the name and address entries as illustrated below.

$$\begin{aligned}&(s_1, h(s_1 \circ \text{name} \circ \text{key}))\,,\\&(s_2, h(s_2 \circ \text{name} \circ \text{ip\_addr} \circ \text{key}))\,,\\&\text{date\_and\_time\_entry\_added}\end{aligned}$$

When a host is contacted in the future, its key will be retrieved before the `known_hosts` file is searched and so it is still quite possible to check whether the key is associated with any known host name/address pairs. Obtaining the keys requires the attacker stage an online dictionary attack, contacting hosts that it may not be able to authenticate to and increasing the likelihood of detection. Passing a large dictionary of these keys around with a worm would be bandwidth intensive and likely to raise alarms.

The additional benefit incurs a significantly higher usability cost than the previous approaches. First, both the host name and the key are required in order to identify or remove an entry from the `known_hosts` database. If a key was lost and needed to be revoked, a revocation list would need to be employed to revoke all keys assigned to that host before the date on which the key was replaced. What's more, users would not differentiate between the response received when they first contacted a host and the response received when a host's key changed. Fortunately, the correct security behavior in both cases should be the same – the user should check the host key's hash against a hash obtained through a secure alternate channel.

While this alternative design will counter *offline* dictionary attacks, *online* dictionary attacks remain a concern, especially if no system is in place to detect them. An attacker staging an *online* dictionary attack can fetch the host key once and test it against each user's `known_hosts` database on the compromised host. To make *online* dictionary attacks less effective, it would be beneficial if the key the client expected from the server changed for each user. This approach would be most acceptable for protocols in which clients and servers, after authenticating in a first communication, agreed upon a symmetric key for future sessions. Alternatively, the user's identity key could be used to create a certificate that the server could use in future communications with the user to assert the authenticity of its key.

6.2 Protecting configuration files

Host address hashing can also be used to protect addresses in user-configured files such as the trusted host file (`.shosts`) and the user's main configuration file, so long as the host name need not be read until the host to be contacted has been identified. However, using incomprehensible tokens in place of plaintext addresses in these files may raise concerns for any sophisticated user or system administrator who may want to audit these files to ensure they do not place trust in the wrong remote hosts.

Fortunately, there is more flexibility in designing solutions to this problem than that of the `known_hosts` database, as configuration files are not written by SSH. Thus, solutions do not need to support mechanisms through which the SSH client or server can change the file.

To ensure that configuration files could be audited, hashing approach (2) could be modified to use a deterministic public key encryption algorithm as its hashing function. While the function remains one way and collision resistant to those without the key, an auditor with the key can reverse the function.

Some may find it simpler to use hashing approach (2) and to maintain an encrypted master configuration file in which host names and addresses are not hashed. To modify the configuration, the administrator decrypts the master file to plaintext, makes changes to this plaintext master, copies it, obfuscates the addresses in the copy, and finally re-encrypts the master file. However, if the file is changed it may not be possible to determine how it was changed.

6.3 Protecting log files

The log files generated by the SSH server not only contain the names of other hosts running the SSH protocol suite, but also the names of the user accounts on those hosts. While this information is dangerous in the hands of a worm, its presence can be essential to detect and track intrusions. Logs should be easily converted back to plaintext form for processing. Fortunately, in exploring the solution space to this problem we can take advantage of the fact that logs need not be written by users or administrators and, more importantly, that they need not be processed by anyone other than the system's administrators.

We can prevent log entries from being harvested if we can encrypt these entries to ensure that, once written, they can only be read using a secret key. A naïve algorithm to accomplish this would encode each entry using a public key cryptosystem. Less computationally intensive approaches to securing audit logs have been introduced by Yee and Bellare [28], Schneier and Kelsey [17], and Waters *et al.* [24].

A simplified algorithm that meets our requirements can be constructed using a public key pair. When the SSH server begins executing, it creates a random session key k_0 for use with a faster symmetric cryptosystem. It then encrypts this k_0 with the public key, encodes the result into base64, and writes it to the log.

Each log entry then begins with its sequence number, i, followed by the entry contents encrypted with symmetric key k_i, where $k_i = h(k_{i-1})$. Once the logging function has encoded the entry, it immediately calculates k_{i+1} and discards k_i from memory. To derive k_i from k_{i+1} would require breaking the one-way hash function.

When the system administrator wants to review the log he must provide his private key, which is password protected and ideally stored on a a host other than the one generating the log. The private key is used to decrypt k_0. The key for any entry i can then be derived by calculating $k_i = h^i(k_0)$.

6.4 Protecting gateway hosts

While hindering attempts to harvest addresses can help to thwart the spread of attacks through gateway hosts, it is preferable to avoid running SSH clients on these hosts altogether. An ideal SSH gateway is one on which the SSH server, but not the SSH client, is installed, and through which users can forward TCP connections but execute no other operations. To initiate a connection from a local client to a server through such a gateway, users first initiate an SSH connection to the gateway and then initiate a second SSH connection from the local client to the server through the gateway. This is one of the approaches recommended in the text *SSH, the Secure Shell: The Definitive Guide* [1].

Unfortunately, the methods available to forward connections through the gateway are less than straightforward and beyond the knowledge and abilities of most users. One method of constructing forwarded SSH connections is to setup a proxy in the configuration file, but this must be created for each gateway and makes the user's configuration file a more attractive target for harvesting. This method also presents problems if the gateway uses any form of interactive authentication, such as host password authentication.

Another means to accomplish a forwarded connection is to use local port forwarding. However, this opens up the gateway to abuse from others on the same

```
myclient> ssh -H lazlo@gateway -H server
Establishing forwarded connection. Be sure to
close this shell window immediately after your
session is complete.

Authenticating user 'lazlo' to 'gateway':
----------------------------------
|| password: ***********        ||
----------------------------------

Authenticating user 'hollyfeld' to 'server':
----------------------------------
|| password: *************      ||
----------------------------------
Connected to 'server'.
server>
```

Figure 4: A forwarded connection using the `-H` option. Boundary boxes surround interactive authentication sessions, ensuring that the gateway host cannot use the session to fool the user into issuing commands intended for the client or server hosts.

client host with access to the forwarded port. This method also requires that the user learn how to use the `HostKeyAlias` configuration option so that the connection forwarded through the local port isn't treated as a connection to localhost in the `known_hosts` database. Finally, the user must use two different shells on the client host to initiate clients and their connections to the gateway and server. If a single shell is used, the gateway can initiate an interactive authentication session during which it spoofs the behavior the user expects after the gateway connection is completed. The user may then end up typing commands (and even passwords) to the gateway while thinking he is sending them to the server.

Given the complexity of the available options, it's little wonder that most users simply issue a command to the SSH client on the gateway host if one is available. To ease the process of constructing forwarded connections, we propose a SSH client command option, which we are currently implementing.

```
ssh -H gateway -H server
```

The `-H` option indicates the start of a new connection in a connection chain. In the above example, the client establishes a connection with the gateway and then uses a forwarded connection to contact the server. Any number of gateway hosts can be used, each of which is contacted using a separate SSH client process. Local port forwarding is performed using UNIX domain sockets to avoid opening TCP/IP ports accessible to other users. Options specified before the first `-H` are applied to all forwarded connections if appropriate. Options specified between a `-H` and a host name are applied only to that connection.

If interactive authentication is used, the authentication interaction is confined within a box that clearly indicates the host to which the user is authenticating as shown in Figure 4. All characters used to manipulate the cursor position are ignored, with the exception of line feeds which cause a new line to be created within the box. If characters exceed the length of the screen a new line is created within the box.

7 Conclusion

We have explored the emerging threat to hosts that rely on SSH for their security and the form in which future attacks may take. In particular, we have articulated eight impersonation attacks on SSH that either exploit misplaced trust, use stolen credentials, or insert new commands or credentials through stolen SSH sessions. Each of these attacks can be exploited by an SSH worm when combined with address harvesting of `known_hosts` databases and other files.

To show the scope of the threat of an SSH worm, we collected data from 92 hosts and located `known_hosts` relationships with 8,009 hosts on 55% of all valid /8 networks. We have also collected evidence indicating that identity keys are, more often than not, stored unencrypted. These facts help to explain why attackers that target SSH have been able to quickly compromise new hosts on new networks.

To address the ease with which host names and addresses of SSH servers can be harvested from the client's file system, we have presented a series approaches for hiding these addresses using hashing. These approaches include countermeasures not only against plaintext harvesting, but also against attempts to guess host names and addresses. Finally, we suggest improvements to existing approaches to forwarding SSH connections through gateway hosts in order to reduce the effectiveness of attacks on these hosts.

8 Epilog

This paper was first conceived in early 2004 and drafts have been in private circulation since June of that year. Only late in the year did we first learn of the profusion of real world impersonation attacks taking place against installations of SSH.

On February 15, 2005 an updated draft was submitted to officials at F-Secure, SSH Inc., and the OpenSSH development team with a notification that public release of this work was imminent.

OpenSSH responded with by creating their own implementation of host address hashing as part of OpenSSH 4.0 on March 9, 2005. Unfortunately, this implementation is turned off by default and does not come with a script with which a system administrator can update all of `known_hosts` files on a system. We have provided such a script and instructions for turning hashing on at `http://nms.lcs.mit.edu/projects/ssh/`.

As of May 10, 2005, F-Secure and SSH Inc. have yet to respond.

9 Acknowledgments

The authors offer our gratitude to all of those who have anonymously contributed data to our efforts at the expense of their own time and effort.

The authors would like to thank Lou Anschuetz, Hari Balakrishnan, Robert Cunningham, David Dagon, Victor Hazlewood, Glenn Holloway, Roger Khazan, Cynthia McLain, David Molnar, Scott Pinkerton, Michael D. Smith, and Bill Yurcik for their advice and comments. Stuart Schechter would like to thank the National Science Foundation for supporting this work while he was at Harvard under grant number CCR-0310877, and the MIT Lincoln Laboratory Advanced Concepts Committee for its support since his arrival at the laboratory.

References

[1] Daniel J. Barrett and Richard E. Silverman. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, February 2001.

[2] Matt Bishop and Daniel V. Klein. Improving System Security via Proactive Password Checking. *Computers and Security*, 14(3):233–249, 1995.

[3] SANS Internet Storm Center. Port Graph (for port 22). `http://isc.sans.org/port_details.php?port=22&days=70`.

[4] David Dagon. Email correspondence, December 10, 2005.

[5] Arthur Evans Jr., William Kantrowitz, and Edwin Weiss. A User Authentication Scheme Not Requiring Secrecy in the Computer. *Communications of the ACM*, 17(8):437–442, 1974.

[6] F-Secure. F-Secure Virus Descriptions: Deloder. `http://www.f-secure.com/v-descs/deloader.shtml`.

[7] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical UNIX & Internet Security*. O'Reilly Media, Inc., Sebastopol, CA, 3rd edition, February 2003.

[8] Victor Hazlewood. Security Technologies Manager, San Diego Supercomputer Center (SDSC), Telephone correspondence, January 18, 2005.

[9] Internet Assigned Numbers Authority. Internet Protocol V4 Address Space. `http://www.iana.org/assignments/ipv4-address-space`.

[10] Internet Assigned Numbers Authority. *RFC 3330: Special Use IPv4 Addresses*. IETF, September 2002.

[11] Michael Kaminsky. *User Authentication and Remote Execution Across Administrative Domains*. PhD thesis, Massachusetts Institute of Technology, September 2004.

[12] Michael Kaminsky, Eric Peterson, Daniel B. Giffin, Kevin Fu, David Mazires, and M. Frans Kaashoek. REX: Secure, Extensible Remote Execution. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 199–212, June 2004.

[13] Robert Morris and Ken Thompson. Password Security: A Case History. *Communications of the ACM*, 22(11):594–597, 1979.

[14] National Institute of Standards and Technology. Secure Hash Standard. FIPS PUB 180-1, April 17, 1995.

[15] Scott C. Pinkerton. Network Solutions Manager, Argonne National Laboratory, Email correspondence, February 4, 2005.

[16] Stuart E. Schechter, Jaeyeon Jung, and Arthur W. Berger. Fast Detection of Scanning Worm Infections. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*, September 15–17, 2004.

[17] Bruce Schneier and John Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[18] K-OTIK Security. SSH Remote Root password Brute Force Cracker Utility. `http://www.k-otik.com/exploits/08202004.brutessh2.c.php`, August 20, 2004.

[19] Abe Singer. Tempting Fate. *;login: The USENIX Magazine*, 30(1), February 2005.

[20] Eugene H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Purdue Univerisity Department of Computer Sciences, 1998.

[21] Symantec. Security response–W32.HLLW.Deloder. `http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.deloder.html`.

[22] Symantec. Security Response–W32.HLLW.Gaobot.AA. `http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.gaobot.aa.html`.

[23] Symantec. Security Response–W32.Lovgate.mm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.lovgate@mm.html`.

[24] Brent R. Waters, Dirk Balfanz, Glenn Durfee, and D. K. Smetters. Building an Encrypted and Searchable Audit Log. In *Proceedings of the 11th Annual Network and Distributed Security Symposium (NDSS ’04)*, February 1–6, 2004.

[25] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the 13th USENIX Security Symposium*, August 9–13, 2004.

[26] Thomas Wu. The Secure Remote Password Protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, March 1998.

[27] Jun Xu, Jinliang Fan, Mostafa H. Ammar, and Sue B. Moon. Prefix-Preserving IP Address Anonymization: Measurement-based Security Evaluation and a New Cryptography-based Scheme. In *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP ’02)*, November 12–15, 2002.

[28] Bennet S. Yee and Mihir Bellare. Forward Integrity for Secure Audit Logs. Technical report, University of California at San Diego Department of Computer Science and Engineering, November 1997.

[29] William Yurcik. Senior Systems Security Engineer, The National Center for Supercomputing Applications (NCSA), Telephone correspondence, January 28, 2005.

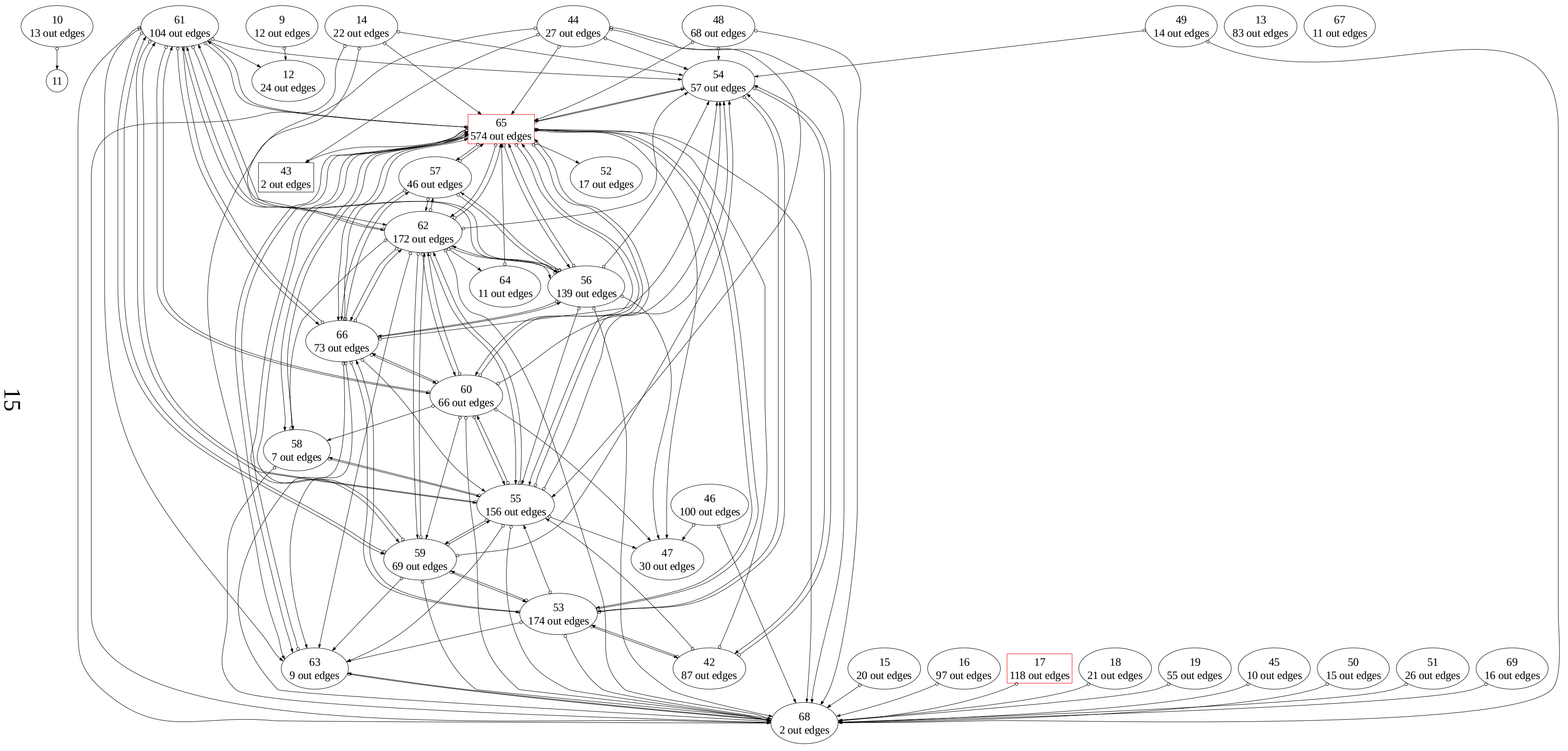


Figure 5: This graph represents 39 hosts within a single institution that submitted `known_hosts` entries. 1251 terminal nodes, those nodes in the network that are destinations of `known_hosts` entries but from which we did not collect entries, are excluded. Rectangular nodes are the ones on which our data collection script ran as root.

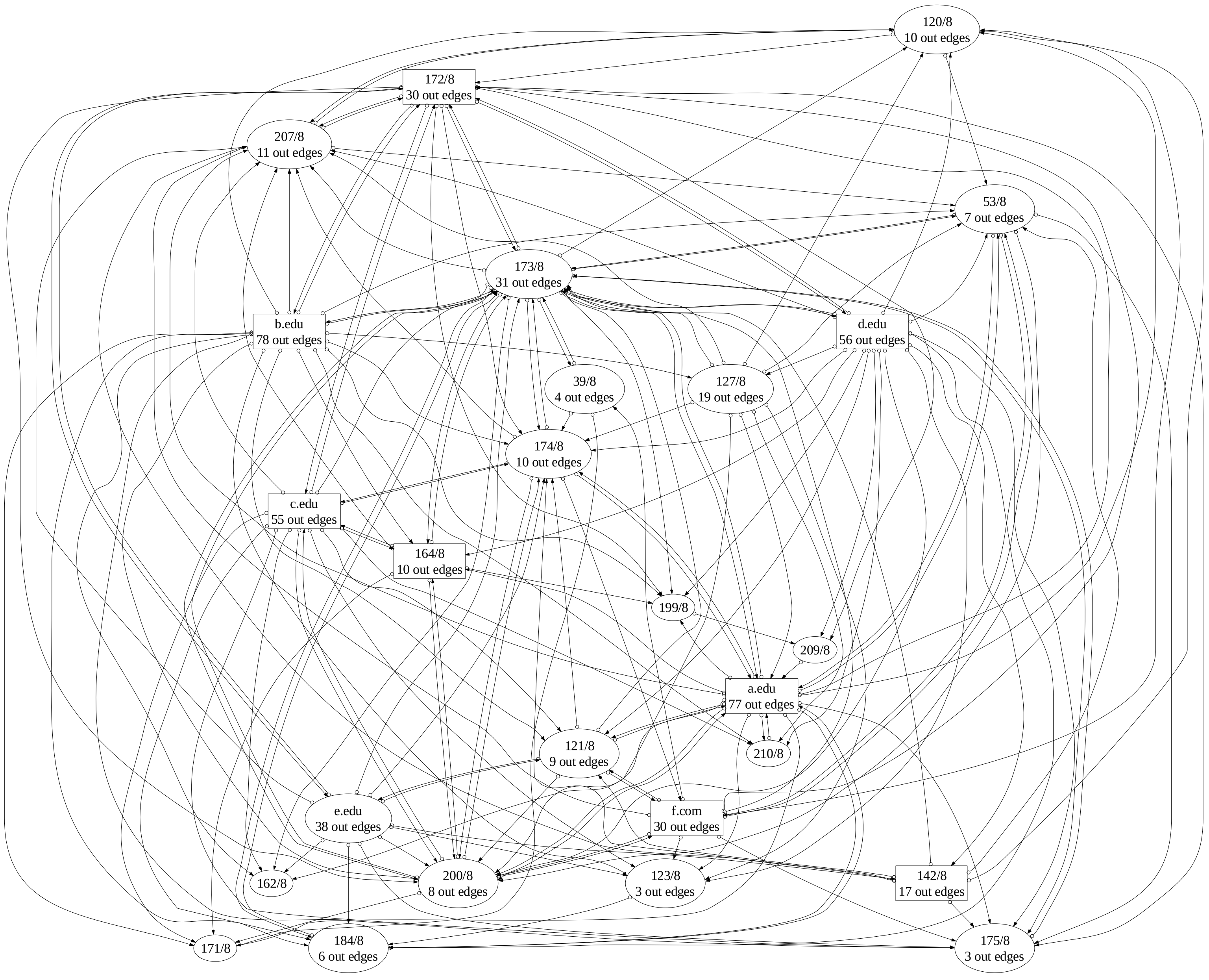


Figure 6: Each of the nodes in the graph represents a host or set of hosts from either an academic institution's network (.edu), business network (.com), or a /8 (Class A) network. A single edge is used to represent all of the `known_hosts` relationships from a source network to a destination network. 70 terminal nodes are excluded. Rectangular nodes those networks from which we were able to collect data from ten or more user accounts.