Camouflaging Servers to Avoid Exploits[†]

Morgon Kanter and Stephen Taylor
Thayer School of Engineering at Dartmouth College
tr11-001

Abstract: *The goal of this research is to increase attacker workload by camouflaging servers. Server vulnerabilities are dependent on the specific operating system or server type, version, service pack, and/or patch level. Protocol definitions offer considerable flexibility to developers, and as a result it is possible to fingerprint a particular server by communicating with it using either legitimate or malformed traffic. This fingerprint information provides a roadmap that allows an attacker to select an appropriate exploit and compromise the server.*

This paper describes a general camouflage capability that presents a false server fingerprint. The capability is implemented as a table-driven finite state machine that operates across the protocol stack, simultaneously falsifying both operating system and service properties. The false fingerprint may be created to provide known vulnerabilities, that if exploited can trigger an alert or honeypot the attacker. The camouflage has been demonstrated by disguising a Microsoft Exchange 2008 server running on Windows Server 2008 RC2 to appear as a Sendmail 8.6.9 server running on Linux 2.6. Both the nmap and Nessus network scanners were deceived into incorrectly identifying the Exchange server. It is important to recognize that camouflage need not be a perfect deception: it is sufficient to sow enough confusion that an attacker is unable to take timely actions.

1 Introduction

Before a known server vulnerability can be exploited, it must first be discovered and a clear picture of the operating environment on the server must be developed. Several different tools are available for operating system and service detection; two of the most popular (1) of these are nmap (2) and Nessus (3).

At their core, these scanners operate due to an inherent issue in protocol specification and implementation: protocol specifications typically leave many implementation details up to the developer. This allows for multiple implementation strategies and ideas to be used in alternative products. When these design details diverge between competing systems, the differences can be observed by merely using the protocol and system fingerprinting is enabled.

Nmap and Nessus both exploit the differences in implementation details for their discovery process. For example, for operating system detection they have databases specifying which details are expected on which system. By examining the response to TCP/IP traffic, they discern which operating system the responses must have originated from. Similar systems exist for application-layer protocols in Nessus (4).

Since service detection is performed by examining implementation differences, it is possible to create deception by modifying these differences Early uses of this idea, implemented in Morph (5), were able to transparently camouflage a system to appear as Windows 2000, OpenBSD, or Linux 2.4. The concept has also been used in FreeBSD, which scrubs its fingerprint so that it is not detectable by scanners (6). Linux 2.4 provides a program called IP Personality that allows it to take on alternative operating system characteristics (7). All of these packages focus on manipulating TCP/IP protocol details to prevent operating system detection.

We extend this work across the protocol stack to include the application-layer protocol. Scanner developers have discovered that detecting TCP/IP idiosyncrasies alone has weaknesses (8). Nessus in 2009 modified their operating system detection to examine application-layer services. This enhancement weighs detected services into the detection procedure along with classic TCP/IP stack fingerprinting (9). Although Nmap currently separates application and TCP/IP fingerprinting, it is likely that future versions will combine them.

The goal of the work described here is to increase attacker workload by camouflaging servers. Most traffic in modern networks is between client and designated servers, rather than client-to-client. As a result, servers represent high-value targets for exploitation. Our approach is based on a table-driven finite state machine that deceives attackers, causing delays, confusion, and the use of inappropriate exploits. The capability we have developed defeats the standard

† This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

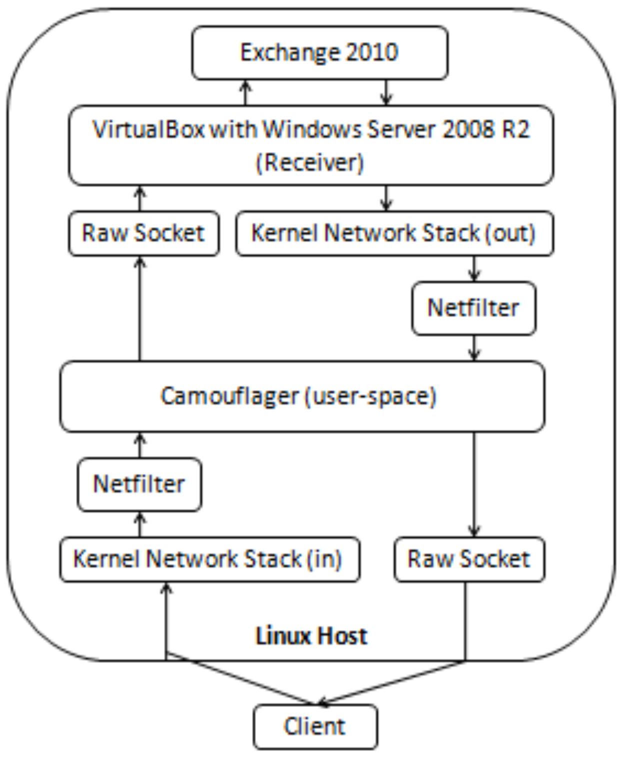
Exchange 2010
VirtualBox with Windows Server 2008 R2 (Receiver)
Raw Socket
Kernel Network Stack (out)
Netfilter
Camouflager (user-space)
Netfilter
Kernel Network Stack (in)
Raw Socket
Linux Host
Client

Figure 1: Overview of System Internals.

database-backed operating system fingerprinting capabilities of both nmap and Nessus. It also deceives nmap's application-layer service detection for SMTP and, if this is used to detect operating systems, its service-based method of operating system detection.

2 Methodology

Figure 1 shows an overview of the camouflager proof-of-concept implementation and its components. The camouflager is implemented as a user-space program running on a Linux host. These mechanisms could also be deployed in a proxy server, router, or, natively in the backend server.

Packets are initially received from a remote client by the host kernel. The Netfilter QUEUE mechanism of Linux allows packets matched by arbitrary iptables rules to be temporarily removed from the kernel's network stack. These packets can then be brought into user-space for modification by the camouflager. The camouflager may also add or remove packets from the stream. All packets are then reinjected into a virtual machine, running Windows Server 2008, through a raw socket. This virtual machine is the backend Exchange 2010 server being camouflaged as Sendmail.

Packets emanating from the Exchange server travel a similar path in reverse. When they reach the Linux host's network stack, they are captured by Netfilter, modified by the camouflager, and reinjected into the network via a raw socket. They then travel to the client, camouflaged so the client will believe it is communicating with a Sendmail server instead of an Exchange server.

The internal details of the camouflager are shown in Figure 2. It is implemented as two separate table-driven finite state machines; one for the application layer and the other for the network layer. The tables used in each finite state machine define a collection of header and payload matches, which then cause substitutions and rearrangements. These changes vary depending on the operating system to be camouflaged. The same camouflage path is used for both client-to-server and server-to-client communication; however, client-to-server packets are never modified; instead they simply change the state of the application layer machine. This is because there is no need to camouflage incoming packets, only the packets originating from the backend server are important in deceiving attackers.

The application layer camouflager cannot act on each packet independently since there are no guarantees that a complete application-layer message is contained in each packet. For example, half of an SMTP command may be contained in one packet and the remainder in the next. Therefore it is necessary to concatenate packets as they arrive to form complete application-layer messages. These messages can then be passed to the application-layer camouflager for modification. The modified messages are subsequently repacked into packets and forwarded to the network camouflager. Since it may be necessary to add or remove packets, the application-layer camouflage must be completed before the network layer. Complete messages are delineated using a regular expression match tested as each packet is received.

After the application-layer messages are camouflaged, the network camouflager examines and modifies packet headers. This operation is based on the detection mechanisms of tools such as nmap and Nessus (10) (11).

2.1 Application Layer Camouflage

Recall that our proof-of-concept camouflage is tailored to SMTP, deceiving an attacker that Exchange is Sendmail. It is sufficient for the finite state machine to operate on a restricted subset of the protocol sufficient for basic email exchange. The camouflager modifies the initial banner, as well as the server responses to the SMTP verbs HELO, EHLO, RSET, HELP, MAIL [FROM], RCPT [TO], DATA, QUIT (12). For example, the HELO verb when followed by a

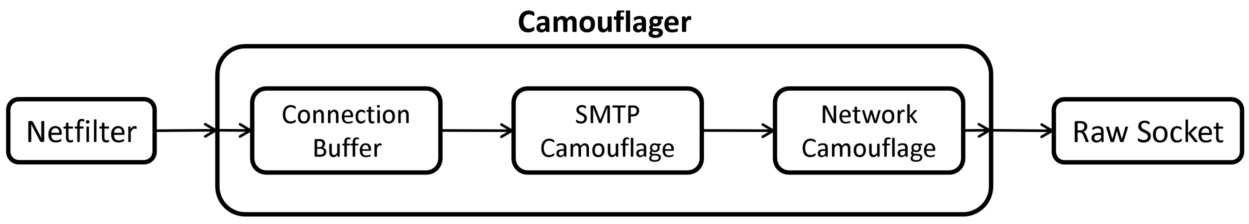

Figure 2: Packet's Path Through the Camouflager.

domain name (e.g. internet.com) is protocol-correct; otherwise, when appearing alone it is incorrect. Both must be accounted for in the camouflage. For Exchange, the protocol-correct response is:

250 <server-domain> Hello [<client-ip>]

To camouflage Exchange as Sendmail the corresponding response is:

250 <server-domain> Hello [<client-ip>], pleased to meet you

For verbs outside the basic set, two modes of operation are supported. In the first, traffic is dropped before reaching the backend server and the response to the client is:

500 5.5.1 Command unrecognized: <client input>

This response indicates that the server is Sendmail and the service is not supported. In addition, the response to the HELP verb produces only the help information for the supported verbs, in order to emulate lack of support for any other protocol options.

Alternatively, the second mode of operation is to forward the command, uncamouflaged, to the backend server and transmit the response unchanged. This alternative simply provides confusing feedback to the attacker.

The typical mail-sending path through the state machine is shown in Figure 3. Upon connection, the server sends the client a banner saying the name and version of the running server. This banner even includes the specific patch level. After the banner, the client sends the verb "HELO" followed by its domain and the server responds with an affirmative "Hello" message. Mail can now be sent from client to server.

To initiate the mail transfer process, the client sends the "MAIL FROM" verb followed by the email address of the sender. The server replies with the code 250, which means that the command was successful. Following a successful "MAIL FROM", the client then specifies the "RCPT TO" verb followed by the email address of the intended recipient. Again, if successful, the server replies with a 250 message. The client then begins transferring the message content by sending the "DATA" verb; the server replies with a 354 message, indicating it is ready to receive the content. The client then sends the content of the email and specifies the end of the message by sending "." on a line by itself. The message is then queued for delivery by the server, and the state machine resets to the WAIT state.

Other verbs supported by the camouflager, but not displayed in Figure 3, are RSET, HELP, and QUIT; all of these require no parameters and may not be used in the DATA state. The RSET verb is used to reset a session, typically it causes the state machine to return to the WAIT state. The HELP verb provides information on the server and does not cause a change in state. Finally, the QUIT verb causes the server to respond with a server-specific "goodbye" message and causes the state machine to close the connection and enter the UNCONNECTED state.

Generally, transitions outside of those shown in Figure 3 that are not associated with RSET, HELP, and QUIT, represent errors in use of the protocol. Generally, an error will always send the following error message to the client:

500 5.5.1 Command unrecognized: <client input>

The camouflager state is unchanged after an error occurs. For more details on the operation of the listed verbs and how they work together to form a complete mail session, see (12).

Although this is only a small subset of the complete protocol, it is sufficient to perform normal email exchange functions. It is also enough to deceive current automated service-detecting systems such as nmap.

2.2 Network Layer Camouflage

The network layer camouflager is a state machine similar to the application-layer camouflager; however, most camouflage operations are applied to all

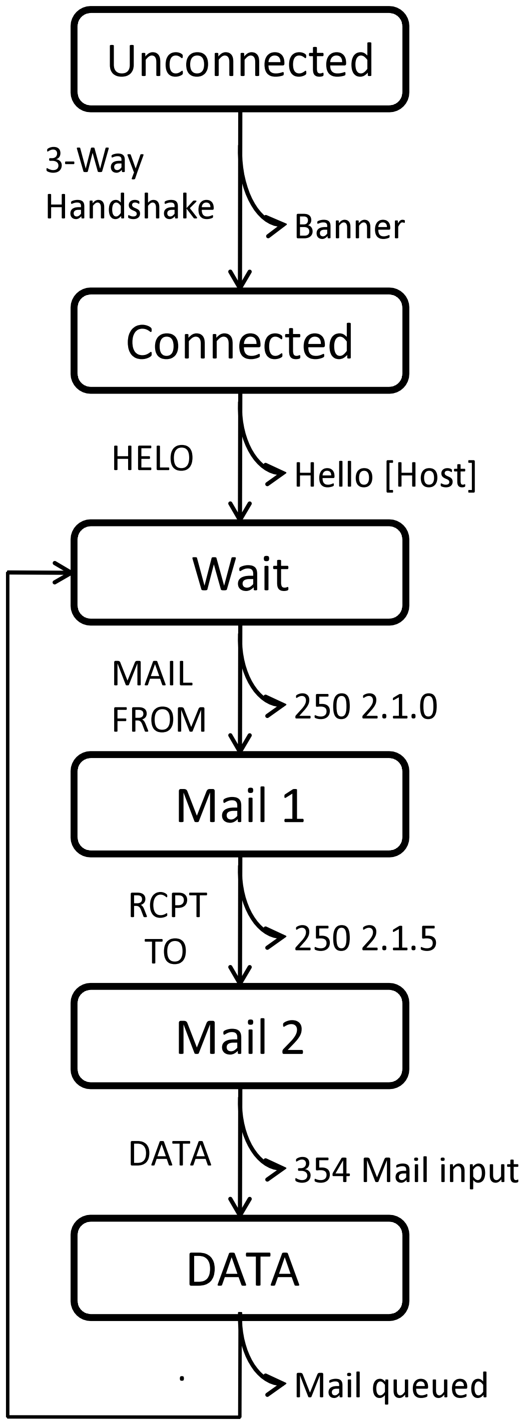

Figure 3: State machine representation of the common path of SMTP camouflage.

packet headers. Very few involve introspection into the protocol operation. The network layer camouflager also handles the creating, updating, and deleting of internal structures containing the application-layer message buffers, state variables, and protocol information. Just as in the protocol itself (13), a connection structure is created on the receipt of a SYN message indicating the beginning of a connection. When this happens, a new structure for storing state variables and buffers is created. If the packet is not a SYN packet and no connection structure can be found (i.e. the client IP and Port are unknown), the packet is injected back into the network stack with no camouflage action taken. Connection structures are deleted when a FIN message and its acknowledgment have been received from both client and server, as per the normal operation of TCP.

Various parts of the TCP and IP header need to be manipulated in order to deceive the network stack fingerprinting methods of nmap and Nessus (10) (11). These parts represent specific idiosyncrasies in the protocol headers that Windows and Linux implement in different ways. There are flags that are set or unset, values given for different parameters, optional arguments present or absent and, finally, the specific ordering in the header of the various optional parameters.

The initial window parameter for TCP packets from Linux 2.6 is often a hexadecimal 16A0, as opposed to Windows Server 2008 where it is a hexadecimal 2000 (14). Likewise, the optional window scale parameter is different – 05 in hexadecimal for Linux 2.6, versus 08 for Windows Server 2008. If the initial SYN has an ECN flag set, the initial window is set as hexadecimal 16D0 instead of 16A0.

The sequence numbers are camouflaged as well – while Nessus does not examine this field, nmap looks in detail at initial sequence numbers provided by the server's SYN/ACK responses (10). It was not necessary to duplicate Linux's initial sequence number algorithm. Instead we generated a random even offset, between 2 and 16 from the last initial sequence number. This has the effect of spacing initial sequence numbers such that greatest common denominator of the differences between consecutive initial sequence numbers is usually between 1 and 6. This is sufficient to distinguish the Linux from Windows on the basis of sequence numbers.

We discovered that Windows Server 2008 R2 does not always respond positively to the ECE flag in TCP used for congestion control, whereas some versions of Linux do. Nmap uses this as part of its fingerprint (10). Thus the camouflager, in response to an ECE packet, responds with the ECE and CWR flags set indicating a positive response (without actually doing any adjustment to the window parameters).

Although it is not tested by either nmap or Nessus, retransmission time is an important metric in TCP stack fingerprinting. When a packet goes unacknowledged, the protocol specifies that it must be resent by the server. The timing of this resending, however, is a detail specified by the system implementation. RING (15) is a tool for fingerprinting the TCP stack of a system based on this retransmission time. The camouflager resends unacknowledged packets based on its own internal timer; for Linux 2.6, we discovered that unacknowledged packets are resent every six seconds by default. Old packets that the backend server resends are dropped because Windows resends packets every sixty seconds.

Finally for TCP, the usage and ordering of options is extremely important for both nmap and Nessus. Nessus's operating system detection is based on SinFP which makes great use of options (11); both ordering and value are part of the nmap fingerprint database (10). For Linux to be positively detected, the following options need to be in the SYN/ACK response, in the following order (16):

- Maximum Segment Size, hex value of 05B4.
- SACK permitted.
- Timestamp (equivalent value to Windows Server 2008 R2 is acceptable).
- NOP x1.
- Window Scale, hex value of 05.

The options and their ordering are different for nmap's ECN probe (14):

- Maximum Segment Size, identical to above.
- NOP x2.
- SACK permitted.
- NOP x1.
- Window Scale, identical to above.

The requirements for the IP header are comparatively simple compared to those for the TCP header. To camouflage a system as Linux 2.6, the "Don't Fragment" bit should always be set to zero, and the TTL when it leaves the camouflager should be set to hexadecimal 3F. The SYN/ACK response to nmap's probes should always have an IP ID value of zero (on Windows Server 2008 R2, it has a random value) (10).

When these changes to the TCP and IP layers are made, nmap detects the camouflaged server as "Linux 2.6.X" with 97% certainty and Nessus always detects the camouflaged server as "Linux Kernel 2.6".

3 Lessons Learned

During the course of this research, several issues emerged as unexpectedly difficult to handle in the camouflage process:

Sequence Numbers. The correct treatment of sequence numbers and acknowledgments emerged as a significant challenge. When modifying application-layer messages, the sequence and acknowledgment numbers sent by the client and the server begin to diverge. For example, if a message sent by the server was originally 15 bytes, in the normal protocol the acknowledgement number will be +15 from the sequence number of the original message. If the camouflager repackages a 15 byte message into 13 bytes, then the relative acknowledgement from the client will be +13 – indicating to the server that 2 bytes were not received and causing a resend.

Divergence is not solely an issue of application-layer camouflage: as mentioned in section 2.2, the initial sequence numbers in the SYN/ACK packets get modified as part of the operating system camouflage, making the sequence numbers between the client and server diverge from the very first communication. However, the divergence due to message size causes retransmission and consequently must be handled by tracking message sizes in the camouflager.

To achieve this tracking and deal with initial sequence number divergence, the camouflager must keep track of the sequence and acknowledgment numbers for both client-to-server and server-to-client communication. It must also modify messages to correct for divergence.

To maintain the sequence number fidelity between both ends of the connection, information about past traffic must be maintained as part of the connection structure. This information includes:

- The current sequence numbers.
- The first unacknowledged sequence numbers.
- For every unacknowledged packet, the sequence number that the sender is expecting to be acknowledged and the sequence number that the receiver will receive and acknowledge.

This information is sufficient to translate between two sequence spaces. Packets that are retransmitted from the server are dropped; as part of the camouflage operation, unacknowledged packets are retransmitted by the camouflager every six seconds to emulate the default network stack functionality of Linux 2.6.

Repackaging. Recall that, while the application-layer camouflager almost always operates on the payload of a single TCP packet, this is not guaranteed and there are times when it requires payloads from multiple packets in order to act. When this happens, the amount of data that needs to be packed in the TCP packets' payloads might be larger or smaller than what was there previously by virtue of camouflage. For normal operation we just divide the data into randomly sized chunks to prevent attackers from detecting camouflage by the use of identically sized packets. In the corner case where such a scheme would push a packet past its maximum length (due to the unlikely normal maximum packet size, or the significantly more likely window maximum) we create and inject into the stream an extra packet. Any timestamp information is copied from the preceding packet in the stream, and the sequence and acknowledgment details in the camouflager are then specifically marked to avoid sending any acknowledgments of the injected packet to the original sender.

Imperfect Command Mapping. Despite the fact that Microsoft Exchange and Sendmail ostensibly implement the same protocol, there are a number of optional components in each program that have no equivalent in the other implementation. An example verb which Sendmail implements, but Microsoft Exchange does not, is VERB[1]. These optional components represent functional differences that cannot be disguised. There are a few options when such cases arise:

- **Pretend that the command worked.** This is a poor choice for any state-based protocol such as SMTP. Take, for example, the verb AUTH, which authenticates a mail sender: not only would the camouflager itself require server knowledge that might not be available to it (such as a database of usernames and passwords), the authentication clearly changes the state on the server. Emulating the ability to use the AUTH command without the state actually changing on the server would result in undefined behavior. This might be a good choice for a verb such as VERB, but it will not work for any command that changes server state. The proof-of-concept camouflager does not support the option to simply pretend that a command was successful.
- **Give a "bad command" response.** This is the approach used by the camouflage program we have developed. This may appear suspicious to an attacker if the server normally supports the command, but it will not result in any undefined behavior. When this option is used, the camouflager loses some of the functionality of the back-end server.
- **Drop the packets.** A similar approach to the "bad command" option, this approach could perhaps emulate a deep-inspection firewall that selectively drops certain commands. The proof-of-concept camouflager does not support this option.
- **Send the message and response through, uncamouflaged.** This causes the camouflager to lose none of the functionality of the backend server, but if the command is specific to the server this will immediately alert an attacker that the server is being camouflaged.

There is, unfortunately, no perfect way to resolve the issue of unsupported functionality. Having the server and camouflager act in a well-defined fashion may leak information that camouflage is present. Even if this happens, however, the actual identity of the end server is still not discernable from this information, provided no messages are sent uncamouflaged. It is important to recognize however, that individual servers may be configured with or without these optional components. Thus the "bad command" response is likely to be interpreted as a legitimate response.

A more complete list of which verbs various SMTP server implementations do and do not support can be found at (17).

Network Stack Issues. Recall that the Netfilter mechanism of the Linux kernel is used for capturing packets. We steal packets with the NFQUEUE target of iptables, which forces the packets to undergo processing in user-space before returning to the kernel network stack. However, due to the limitations in application-layer camouflager, where we might need to forge additional packets as part of the stream, we do not modify the stolen packets and allow them to continue through the kernel network stack. Instead, we copy the packets to user-space and immediately drop them from the kernel's network stack, make our changes to the new packets, and then emit them out of a raw socket so they start anew through the routing tables. This gives us the flexibility to inject new packets into the stream as required.

At first, it might seem like the packets would be queued twice. However, Linux has a marking mechanism, where a sender can mark a packet with an integer number (the default mark for every packet is zero). We mark every packet sent with the number one, and have the Netfilter rules simply ignore every packet that has a mark of one.

The camouflage program also provides a basic proxy mechanism. An incoming packet has its source and destination rewritten so it appears, in both directions, to originate from the camouflage machine. This was developed to simplify routing issues from virtual machines. Sometimes RST packets will be generated by the network stack hosting the camouflage program as TCP packets from both ends pass through it without having an established connection; these RST

[1] Not to be confused with the protocol commands known as "verbs", VERB is itself a verb that enables "verbose mode".

packets must be caught and dropped.

Finally, we discovered that Linux does not always pass certain packets as-is when sent via a raw socket. The response packet to nmap's sixth TCP SYN probe, modified by the camouflager as described in section 2.2, is itself modified by the Linux kernel. Despite use of a raw socket, the IP ID is edited by the kernel for this single packet. For more information on this issue, see (18). To resolve this issue, the camouflage simply drops the response resulting in a 97% nmap identification confidence level, rather than a 100% confidence level.

4 SMTP Session Example

This section presents a standard SMTP session and the differences in application-layer messages that appear when using Microsoft Exchange with and without camouflage. The commands sent are marked in **bold**, and this example presents the mail sending path seen in the state machine in figure 3.

Banner for domain charizard:

Without camouflage:

220 charizard Microsoft ESMTP MAIL Service ready at Wed, 10 Feb 2010 10:58:18 -0800

With camouflage:

220 charizard ESMTP Sendmail 8.6.9/8.6.9; Wed, 10 Feb 2010 11:33:05 -0800

HELO internet.com

Without camouflage:

250 charizard Hello [192.168.56.1]

With camouflage:

250 charizard Hello [192.168.56.1], pleased to meet you

MAIL FROM: <a@internet.com>

Without camouflage:

250 2.1.0 Sender OK

With camouflage:

250 2.1.0 <a@internet.com>… Sender ok

RCPT TO: <b@charizard>

Without camouflage:

250 2.1.5 Recipient OK

With camouflage:

250 2.1.5 <b@charizard>… Recipient ok

DATA

Without camouflage:

354 Start mail input; end with <CRLF>.<CRLF>

With camouflage:

354 Enter mail, end with a "." on a line by itself

Next we enter "abcdef", followed by a carriage return and line feed, then a period, then a carriage return and line feed.

Without camouflage:

250 2.6.0 <9157afd8-b9d7-4390-8034-0f49e017b3f4@charizard.charizard.com> [InternalId=1] queued mail for delivery

With camouflage:

250 2.0.0 o1AGcLfJ001019 Message accepted for delivery

RSET

Without camouflage:

250 2.0.0 Resetting

With camouflage:

250 2.0.0 Reset state

QUIT

Without camouflage:

221 2.0.0 Service closing transmission channel

With camouflage:

221 2.0.0 Service closing transmission channel

This sequence is sufficient to cover the majority of typical mail transactions, and deceive scanners such as nmap and Nessus.

4.1 Generating Results

This example shows how the camouflage deceives the Nmap network scanner for both operating system and service detection. As an initialization step for the camouflage, iptables must be configured with the following rules:

- iptables -A INPUT -p tcp --tcp-flags RST RST -j DROP
- iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP

These two rules configure the raw socket proxy server implementation so that it does not generate any RSTs from the Linux kernel itself. This also serves to block nmap RST probes. Two additional rules are required to catch the incoming and outgoing packets to the SMTP server and copy them to user-space with the Netfilter queue mechanism:

- iptables -A INPUT -p tcp --sport 25 -m mark ! --mark 1/1 -j NFQUEUE
- iptables -A INPUT -p tcp --dport 25 -m mark ! --mark 1/1 -j NFQUEUE

As previously mentioned, outgoing packets are marked with a value of one to avoid capturing them twice.

It is desirable to prevent nmap ICMP probes from reaching the camouflager. Since the camouflage is running on a Linux system, these probes would simply increase the probability of detecting Linux and give a false impression of the effectiveness of the camouflager. Thus the following rule:

- iptables -A INPUT -p icmp -j DROP

The final rule corrects an unwanted behavior present in Linux 2.6.34 (18). This involves the response to nmap's final SYN probe, which has its IP ID rewritten to a nonzero value after sending it to the kernel, even though the camouflager sends it with a zero ID.

- iptables -A OUTPUT -p tcp --tcp-flags SYN,ACK SYN,ACK -m u32 ! --u32 "2 & 0xFFFF = 0" -j DROP

After these rules have been defined, nmap may be invoked with the following command line on another networked computer:

```
nmap -p 25 -O 192.168.1.107 -dd –vv
```

The results identify Linux 2.6.X, with 97% confidence. To demonstrate the effectiveness of the camouflage against nmap's service detection the following command was issued:

```
nmap -p 25 -sV --version-all 192.168.1.107 -dd -vv
```

The results identify "Sendmail 8.6.9/8.6.9", with host charizard and OS Unix.

Use of Nessus resulted in the identification of the camouflaged server's operating system as "Linux Kernel 2.6".

5 Future Work

The proof-of-concept implementation described here demonstrates that application layer camouflage can be achieved and is an effective addition to the arsenal of camouflage options for denying surveillance. It became clear, however, that creating hand-crafted state machines for camouflage was not an effective long-term strategy as it is manpower intensive and costly. Automatically generating the tables, using machine learning methods, from arbitrary traffic streams represents a significant research challenge, but would allow the concepts to be broadly applicable.

There are many interesting opportunities that emerge when a camouflage system is paired with exploit detection. Suppose a server is camouflaged to appear as a server with well-known vulnerabilities. This could entice an attacker into using a known exploit, which could then be easily detected. We have demonstrated this concept already, using a simple intrusion detection sensor and an open-source exploit against Sendmail. It is possible to ignore the exploit, give a bad command message, send an alert to an administrator, or continue operating in a manner that causes further deception and wastes additional attacker resources.

Another interesting direction is to appear as if the exploit succeeded and provide an associated honeypot. This keeps the attacker engaged and away from the actual target longer. Depending on the quality of the honeypot, the attacker may remain engaged indefinitely.

6 References

1. Top 100 Network Security Tools. [Online] 2006. http://www.sectools.org/.

2. *InSecure.org.* [Online] http://www.insecure.org.

3. *Tenable Network Security.* [Online] http://www.nessus.org/nessus.

4. SMTP Server Detection. [Online] Tenable Network Security. http://www.nessus.org/plugins/index.php?view=single&id=10263.

5. Morph OS fingerprint cloaker. [Online] Syn Ack Labs. http://www.synacklabs.net/projects/morph/.

6. *Defeating TCP/IP Stack Fingerprinting.* **Smart, Matthew, Malan, G. Robert and Jahanian, Farnam.** Denver : Proceedings of the 9th USENIX Security Symposium, 2000.

7. *IP Personality.* [Online] http://ippersonality.sourceforge.net/.

8. **Berrueta, David Barroso.** A Practical Approach to Defeating Nmap OS-Fingerprinting. [Online] 2003. http://nmap.org/misc/defeat-nmap-osdetect.html.

9. **Gula, Ron.** Enhanced Operating System Identification with Nessus. [Online] Tenable Network Security, February 2009. http://blog.tenablesecurity.com/2009/02/enhanced_operat.html.

10. **Lyon, Gordon.** Remote OS Detection. [Online] 2010. http://nmap.org/book/osdetect.html.

11. *SinFP, Unification Of Active And Passive*

Operating System Fingerprinting. **Auffret, Patrice.** s.l. : SSTIC, 2008.

12. RFC: 2821. [Online] April 2001. http://www.ietf.org/rfc/rfc2821.txt.

13. RFC 793: Transmission Control Protocol. [Online] September 1981. http://www.ietf.org/rfc/rfc793.txt.

14. nmap-os-db. [Online] [Cited: August 15, 2010.] http://nmap.org/svn/nmap-os-db.

15. **Beardsley, Todd.** Ring out the old, RING in the New. [Online] Plan B Security, May 8, 2002. http://www.planb-security.net/wp/ring.html.

16. **Auffret, Patrice.** SinFP Database. [Online] http://www.gomor.org/bin/view/Sinfp/DocOverview.

17. **McDougall, Wayne.** ESMTP Keywords and Verbs (commands) Defined. *Fluffy the SMTPGuardDog - spam and virus filter for any SMTP server.* [Online] http://smtpfilter.sourceforge.net/esmtp.html.

18. **Kanter, Morgon.** PROBLEM: raw sockets rewriting IP ID in rare cases. [Online] August 13, 2010. http://www.spinics.net/lists/netdev/msg137860.html.