Software Distribution Malware Infection Vector

Felix Gröbert, Ahmad-Reza Sadeghi, Marcel Winandy
Horst Görtz Institute for IT Security
Ruhr-Universität Bochum

Abstract

Software distribution and usage over the Internet has become an integral part of our daily life. On the one hand this is an efficient way to make software widely available to users, particularly for the open source and free software. On the other hand it bears the risk of infecting computer platforms by malicious software since today many software applications are downloaded and installed without appropriate security measures. This situation can obviously be exploited by cyber criminals, and also by Governments intending to deploy spyware against suspects.

In this paper we present an efficient mechanism as well as the corresponding reference implementation for on-the-fly infecting of executable code with malicious software. Our algorithm deploys virus infection routines and network redirection attacks, without requiring to modify the application itself. This allows to even infect executables with a embedded signature when the signature is not automatically verified before execution. We briefly discuss also countermeasures such as secure channels, code authentication as well as trusted virtualization that enables the isolation of untrusted downloads from other application running in trusted domains or compartments.
Keywords: Malicious code, infection, virtualization, Trusted Computing

1 Introduction

Today, software applications (tools, plug-in, etc.) are increasingly downloaded over the Internet making this media an effective means for software distribution. However, software distribution in an open and complex system such as Internet is also exposed to threats like malicious software that exploits the vulnerabilities of today's commodity computing platforms (in particular common operating systems) to gain control over the victim's computing platform or spy on it. The financial losses caused by malicious software are enormous and malware is one of the most severe threats to information society. Hence, potentially every single download could have been infected during the transmission under certain circumstances. Particularly, for the free and open source software it is not common practice to apply corresponding security measures to downloads before executing them. Prominent examples are the browser Firefox and the media player VLC[1].Typical protection mechanisms such as integrity verification and code signing are not applied to the major part of open source application software. In addition new malicious software might not be detected by common anti-malware tools. Moreover, the required organizational and technical infrastructure (e.g., for trust management) either does not exist or seems to be not easy to establish. To infect downloads only an intermediate network node in the chain of nodes from the client to the download server has to act maliciously on the traffic. The network link can be compromised in different ways, e.g., by malicious administrators, a compromised router, or network redirection attacks. This threat gets a new flavor and even more challenging to face when Governments attempt to exploit malicious code[2]. Infection techniques similar to those presented in this paper can be considered for this purpose.

In this paper we present an efficient mechanism as well as a reference implementation for on-the-fly infecting of executable code with malicious software. Our algorithm exploits virus infection routines and network redirection attacks, which will be elaborated in Section 2. Our attack vector simply adds a malicious payload (malware) to the original application during transmission. There are various techniques to infect a binary executable application with malware (see, e.g., [24]). Most of these techniques typically modify the original application to include a small malicious code. In contrast we use a binder (also called companion or joiner) to execute the malicious code along the original application. The infection method of the binder just concatenates the binder application, the malware and the original application. Since original application is not modified one has the advantage that the malicious code can be of a larger size, and thus provide more functionality. Then, upon starting the infected application the binder is started. It parses its own file for additional embedded executable files, reconstructs and executes them, optionally invisible for the user. Another advantage of the binder technique is that the adversary does not need to buffer the file because the

[1] Firefox has 277.000 downloads per day since February 2004 (`http://feeds.spreadfirefox.com/downloads/firefox.xml`), and VLC has an average of 100.000 downloads since February 2005 (`http://www.videolan.org/stats/downloads.php`)

[2] The recent public debate in Germany on the so-called 'federal Trojans' is a good example in this case. The malware is supposed to be developed by the federal investigation office to enable law enforcement to search and spy on the computer platforms of suspects.

adversary just prepends the original file to the binder and the malware. Other infection techniques modify the original file and thus require seeking forward and backward inside the target file. This yields a buffering delay compared to the binder technique. Such a delay might rise suspicion by the victim or may render the attack useless if the victim manually executes additional file checks, i.e. hash comparison with a authentic reference hash. Under certain circumstances this allows for even infecting signed and encrypted executables. If the underlying computer platform does not automatically verify the signature of an executable file before launching it and if it is left to the user to verify it manually. In this case the attack would be successful, since an average user usually does not check whether a binary executable is signed or not.

To mount the attack, the adversary has to infect the binary executable in transmission and at the same time maintain the correctness of the network transmission protocol used for the download, e.g., `HTTP`. Otherwise the protocol parser of the receiving network node will not accept the infected application as a result of the malformed protocol.

Outline: The remainder of this paper is organized as follows: In Section 2 we consider relevant related work. Section 3 presents the adversary and threat model, while Section 4 provides some possible countermeasures against the threats. Section 5 describes our reference implementation of the attack and Section 6 concludes with a discussion and outlook.

2 Related Work

There is a large body of literature on network flow redirection attacks on the multiple ISO/OSI layer 2 and 3 protocols. For the most commonly used protocols one can refer to `802.11` Wireless LAN takeovers [34], `ARP` injection to reroute traffic [33], `DNS` poisoning to insert fake IP addresses for domain names into nameservers [30], `ICMP` redirection messages [15], `BGPv4` route manipulation attacks [17], `IPv6` traffic rerouting attacks [10, 32].

Virus infection routines can be found in many practical tutorials and in textbooks. In [16] a good overview of the `win32 PE`[3] file format is given and different infection methods are outlined. Additional infection methods are presented in [24]. A more formal view can be found, e.g., in [31].

In [22] the authors demonstrate the man-in-the-middle toolkit `ettercap` and mention the possibility of infecting binary executable applications on-the-fly during the download phase. In [27] the author states that a software update server is a significant target, but does not combine this threat with on-the-fly infection routines and network flow redirection attacks.

In [8, 9] security flaws of router firmwares are presented and the author exploits the flaws to modify the router and add an on-the-fly infection routine for `win32 PE` files downloaded through `HTTP`. Their infection method differs from our approach in the following aspects: firstly, it is limited to add a downloader to the binary file by the modification of the DOS stub[4] and `PE` header. The downloader then downloads a specific malware and executes it alongside the original application. We present an on-the-fly infection method, the binder technique, which already binds the malware to the original application and thus overleaps the process of downloading the malware in a separate malicious connection. This technique is more flexible for the adversary. The second difference to our approach is that since our method does not require to modify the original target application file one can attach malware even to an executable with a embedded signature and still succeed to execute the malware under certain circumstances. Thirdly, while we underline the usage of network flow redirection attacks to gain control over the download link, their approach is limited to the modification of router firmware.

3 Adversary and Threat Model

The goal of our attack is to execute malicious code on a target network node. When the target is downloading a binary executable application, our method infects the application during transmission at which we assume that one network node is under control of an adversary (Figure 1). The infection routine adds malicious code (malware) to the binary executable application. Thus, the malicious code will get executed by executing the infected application. We do not discuss detectability or features of malware and malware design, and focus on the infection vector, which can be used in different ways. On the one hand, the adversary can mass exploit multiple victims which may request binary executable applications. On the other hand, the adversary can select single victims and infect few applications. A mass exploitation of the attack vector might be suitable to build up a short term botnet. A selection of a few victims is more suitable to lower the chance that the malware "signature" is detected and included in antivirus tools. For launching the attack we make the following assumptions:

Modification of target traffic. In order to infect a bi-

[3] Windows 32 Bit Portable Executable

[4] A small DOS program to warn the user that the application only runs under Windows

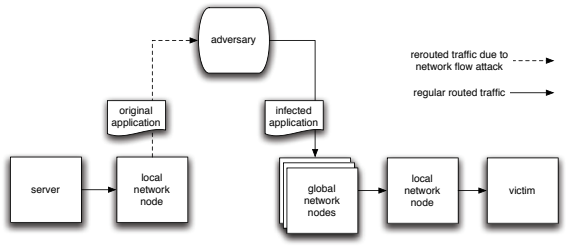

Figure 1: Malicious network node is inserted into the network link using a network flow redirection attack.

nary executable, the adversary has to have the ability to modify network traffic that contains the application. This implies that the adversary is in control of at least one network node which is part of the network link over which the victim is downloading the file. For this, the adversary has various options of choosing between classic attacks to gain administrative privileges on a network node (e.g., buffer overflows) and network flow redirection attacks to include an additional malicious network node within the corresponding network link (Figure 1). A network flow redirection attack, cf. [10, 15, 17, 29, 30, 32, 34, 33], can redirect the traffic to a host over which the adversary has full control. The third possibility concerns a malicious node that may be under the control of a system administrator or that the administrator was tricked to include the node by social engineering. The fourth option can be the law enforcement pursuing a suspected criminal. The law enforcement is able to force an Internet service provider to deploy a "malicious" node inside the network link of certain users.

Absence of protocol inherent integrity checks. The protocols containing the target traffic running between the user and the download server do not enforce encryption or integrity verification of the traffic. The protocol may calculate checksums and add them to protocol packets, but only under the condition that the adversary, i.e., the malicious node, is able to recalculate and adjust the checksums.

Absence of local file integrity verification. The integrity of the file containing the application is not cryptographically checked after the transmission using cryptographic keys distributed a priori, e.g., a comparison of hash of the downloaded file with a trusted database of hashed files obtained over a secure channel. Such checks must not occur manually by the user, by the software distribution system or by the operating system. Due to the nature of the binder (see Section 3.1) the original application may embed signatures and decryption algorithms: the binder reconstructs the original application file and creates a process from the original file. Thus the application will then be in the same state as if the user had ran it manually.

Detectability. Potential antivirus scanners and host or network intrusion detection systems do not detect the binder or the malware. This assumption is reasonable under the condition that the malware deployed by our attack vector is used in a small scale. Mass exploiting a large network with one single malware would lead to detection after some time.

Note that, although the assumptions on the absence of protocol or local integrity verification seem to be very restrictive, this lack of security reflects common practice in commodity operating systems for PC platforms and especially in the context of the distribution of free and open software applications.

3.1 High-Level Construction

The attack vector consist of a set of actions which modify the original application in transmission to deliver a payload functioning as a malware application. There are multiple techniques to infect a binary executable application [24] with malware. Depending on the operating-system-specific binary executable file loader interprets the data structures in the binary file. A binary executable file consists of several sections and a header [16], thus the infection routine first has to determine the ideal position for the malware code inside the original application. An infection routine can add new sections to the application, enlarge sections and append or prepend the malicious code, or find several cavity spaces to spread the malicious code inside the application. At some stage the code has to be executed. Therefor the pointer to the entry code instruction and pointers to other code sections have to be modified. Other techniques use hijacking API function pointers or insert jump instructions which jump to the malicious code in specific places. All of the above techniques have in common that the original application file is modified.

In contrast our approach uses a binder to execute the malicious code along the original application, as illustrated in Figure 2. The binder extracts and executes multiple binary executable files from the infected application file. Optionally, the binder decrypts itself and attached applications, or it may also execute certain malware applications in an invisible way for the user. It has the advantage that the malicious code can be a malware of large size, and thus have more functionality while the original application file structure is not modified and rebuilt before launching.

As the binder technique does not modify the original application, it is well-suited for executables with embedded signatures or code decryption algorithms under the following circumstances: As the original file is reconstructed and executed upon running the binder, the ap-

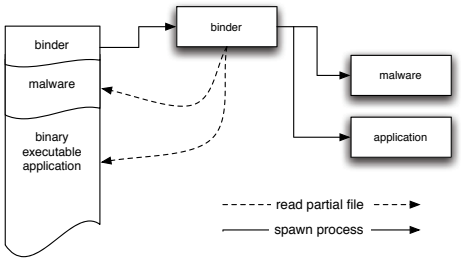

Figure 2: Binder spawning child processes

plication file is in a integral state, i.e., the same state as if it has not been modified, and the application has to specifically search for the binder to determine if a malicious payload was executed alongside the application file. Moreover, there are buffering advantages compared to other infection techniques since they add new sections to a file and modify headers based on those new sections. This requires seeking forward and backward inside the target file, which yields a buffering and computational delay compared to the binder technique. Such a delay might rise suspicion by the victim and may yield the attack unsuccessful if the victim would be able to compare a hash of the file with a trusted reference value in a secure manner, and chooses not to execute the file. The binder just concatenates the binder application, the malware and the original application.

The attack has a simple structure shown in the following algorithms. The contents of the protocol, which is used to transfer the binary executable, are combined to a data stream input to the algorithm 1 `fixprotocol()`. The data stream is read block-wise and the block size and the corresponding logic depends on the underlying protocol. The incoming protocol stream is adjusted, also depending on the protocol, e.g., the checksum or length fields are modified to contain the infected binary executable and at the same time maintain protocol validity. The protocol fields have to be adjusted because a malformed protocol packet would not be accepted by the victim, who would discard the packet, rendering the attack useless. When algorithm 1 `fixprotocol()` has adjusted the protocol fields and a binary executable is detected on the incoming protocol stream `infect()` is called on.

First, `infect()` validates whether (A, B, M) are known binary executables of the same type and target operating system: `infect()` buffers 512 bytes from the stream A and the files B and M and compares the data structure to known data structures of binary executables. This ensures that the malware and the binder can be executed and are functional on the victim's operating system. If there a mismatch is found, `infect()` discards

Algorithm 1 Protocol Handling Algorithm: fixprotocol()

Require: Incoming and outgoing protocol data stream P and $P_{infected}$, binder B, malware M

Ensure: P contains a protocol without cryptographic integrity checks, authentication or confidentiality

```
while P_block ←read incoming logic block from P do
  if P_block = protocol header then
    P_block−infected ←fix protocol headers, i.e., adjust length fields to include sizes of B and M
  end if
  if P_block = protocol checksum then
    P_block−infected ← fix protocol checksum to include B and M
  end if
  if P_block = binary executable application then
    P_block−infected ← infect(P, B, M)
  end if
  send P_block−infected to outgoing protocol data stream P_infected
end while
```

Algorithm 2 Infect Binary Executable Application: infect()

Require: original and infected binary executable application data stream A and A_{inf}, binder B, malware M

Ensure: A, B, M are binary executables of the same operating system

```
A_inf ← B||M
while A_block ← read byte from A do
  A_inf ← A_block
end while
```

the application. If not, `infect()` concatenate the binder B and the binary executable malware M and prepends the result to the original application A. The result is denoted by A_{inf}, which is the output data stream and returns the bytes to `fixprotocol()` for further protocol handling.

4 Countermeasures

In this section we consider some countermeasures against infection attacks.

Network Integrity Assurance. A short-term countermeasure is the usage of security protocols that deploy cryptographic measures. One possibility is VPN solutions like OpenVPN [2] or `IPSec`, which aim at providing integrity, authenticity and confidentiality of the message payload containing the downloaded binary executable file. Another possibility is to use end-to-end cryptographically secured protocols, e.g. `HTTPS` [18], which also aims at assuring integrity, authenticity and confidentiality. In this case the adversary would have to launch a man-in-the-middle attack on `HTTPS`. Such an attack yields a warning window in current browsers,

that the server certificate is invalid, i.e., unsigned by an certificate authority. But certain browsers contain client-side flaws which enable the spoofing of the integrity of a SSL certificate [20]. If we consider VPN solutions, we have to keep in mind that a VPN only secures a subset of the whole network link, over which the binary executable application is transmitted. The other side of the network link remains insecure. An attacker, who has control over a local network node on the side of the server, or a global network node, would still be able to modify the traffic.

File Integrity Assurance. If the network protocol was unable to detect the modification of the binary executable application file, software that is already installed has to detect the modification of the received file. In a binary signing and verification system, executable files are cryptographically signed and the signature is inserted as a specific data-structure inside the file itself. If the application is started the OS can verify the signature and optionally inform the user. We can deduce that the binder method also works in this case. When the infected application, e.g., the binder, is launched, it is started in a unsigned context. Although the unsigned context prior or after the launching of the application might raise suspicion of the security-aware user, it is common that most applications are unsigned. Furthermore on Windows XP the user has to manually check the signature of the file, whereas Windows Vista checks the signature automatically before execution. The binder then extracts the original and signed application, and starts it. Upon starting the signed application, the binder has reconstructed the original state of the underlying executable file, and the OS loader will determine that the signature is valid. Hence, to avoid this situation the signature of the downloaded file must be verified before launching the file, i.e. the binder.

There exist OS approaches that verify executable files before loading and running them. For instance, TrustedBox [19] is a modified Linux kernel which prohibits the execution of modified program files. However, the verification is only performed in a special trusted state whereas the system has to be booted from a non-writable boot media which contains the reference values for integrity check. Thus, approaches like this are not suitable for downloadable applications where the reference values are not known a-priori. In [21] the kernel prohibits execution of modified programs. It identifies unauthorized modifications based on verifying cryptographic signatures which are stored with each application file. However, this approach requires each application to be digitally signed and the secure distribution of signature verification keys. The problem of secure key distribution is out of the scope of this paper. A common misconception is that the trivial file check using the comparison of hashes, shown on a HTTP website, does not verify the authenticity since the reference values would have to be obtained through a trusted channel.

Detection. There are various approaches towards malware detection [12], however, if one makes the assumption that the malware, which is delivered along the binder, is not detected, the only possibility for antivirus scanners to detect the attack is to detect the signature or the behavior of the binder (Section 5). This is along the lines of the typical arms-race in the virus vs. antivirus world today. Additionally, the binder can be rewritten and compiled in ways that a current signature could not detect it. Furthermore, the binder can use techniques like metamorphic[5] code [23] to evade virus scanners. Also, the usage of another infection technique must be considered [16].

Trusted virtualization. To consider the problem of infection more fundamentally, a promising (and recently rediscovered) approach is towards secure and interoperable operating systems. Commodity operating systems do not provide the properties required to reduce the impact of malware due to lack of appropriate security functionality on the one hand and due to their complexity on the other hand. Moreover, the compatibility to legacy systems is an important requirement for any new system to be widely deployed[6]. In this context virtualization technology provides an efficient means for isolating potentially critical applications from others while allowing the interoperability and re-use of existing operating systems and applications. Virtualization is an already known and probed technology and today it is supported by the new processor generation [11] and [6]. It is rediscovered recently, particularly in the context of Trusted Computing. The combination of virtualization and Trusted Computing technologies provides a set of security features such as secure/verifiable boot, and isolation and controlled communication between virtual machines. Some examples in this context are [28, 7, 25, 14, 13]. More concretely, [13] gives an implementation based on trusted computing functionalities according to specifications of the TCG (Trusted Computing Group) and a small security kernel aiming at integrating and enhancing solutions based on identity providers (like password managers [7]), and also at providing protection against malware and interface spoofing like picture-in-picture attacks[7].

[5] Code that mutates itself while keeping its intended algorithm intact

[6] This aspect is a major reason why many secure operating systems developed during the past 25 years have not been commercially successful.

[7] See www.emscb.org

Anonymizers. Anonymizers[8] like Tor [4] can be deployed to support the mentioned means to hide identities in case Governments deploy infection mechanisms to trace suspects. Note that the state can be considered as a very powerful entity with access rights to many resources. Typically, law enforcement targets a limited number of suspects, thus we can assume that law enforcement has to identify the victim before the attack. However, this would be a hard task if the suspects use anonymizers.

5 Implementation

Our proof of concept implementation consists of two parts: `cyanid` and `calcium`. `Cyanid` is a toolchain (based on [1, 5, 3]) to fetch, filter and modify `HTTP` downloads, and `calcium` is a binder to infect `win32 PE` binary executables. The implementation allows for adding a chosen malware, e.g., a Trojan, to a binary executable `win32` application which is downloaded via `HTTP` by the victim. The malware is executed in a hidden form, using the Windows API, and in parallel to the original application. The target has to run Windows 98 or newer.

Proxy. The `cyanid` proxy toolchain uses existing software to modify `HTTP` binary downloads. The toolchain consists of Netfilter [1], Transproxy [5] and a patched Privoxy [3]. The Privoxy patch consists of 27 lines of added and modified C code. In analogy to Section 3.1, `cyanid` implements the algorithms `infect()` and `fixprotocol()`.

Using Netfilter and Transproxy we are able to place an intercepting ISO/OSI layer 7 proxy between any `TCP` connection which is routed over a network node we control. Netfilter is a framework in the Linux 2.4.x and 2.6.x kernels to support packet filtering, network address and port translation and other packet mangling. We use the Netfilter `REDIRECT` target to redirect specific `TCP` connections, i.e., with destination port 80, to our localhost. On localhost the Transproxy software listens for incoming connections and forwards the connection including the original destination host to the layer 7 proxy, Privoxy.

Privoxy is a `HTTP` proxy designed to modify traffic. It enhances privacy on the client-side, e.g., by filtering `HTTP` cookies, and modifies websites on the server-side, e.g., by filtering advertisements. Thus, Privoxy can be used to filter and rewrite `HTTP` requests and responses. In order to modify binary data transmitted over `HTTP`, we had to patch Privoxy: we enabled a filter rule notation to specify binary data (`\xNN`) and modified the filter rule structure to include `NULL` bytes in the filter regular expressions. Additionally, we had to enlarge buffer sizes to handle binary data of several megabytes. Then, in the privoxy configuration a filter rule of the type `s-^MZ-\x4d\x5a\x00... MZ-` is specified. The expression `s-^MZ-` matches a `win32` executable file by searching for its characteristic byte sequence `MZ`. These bytes are replaced by the byte sequence of the binder concatenated with the malware. The last two bytes in the filter rule (`MZ`) reconstruct the substituted executable signature. This rule results in the following modified binary file: ((byte sequence of (binder ‖ malware)) ‖ (original binary file)). Because the victim uses a unauthenticated protocol, e.g. `HTTP`, we are able to modify the contents of the protocol.

Binder. The `calcium` binder implementation parses itself for additional executable binaries, extracts and executes the binaries (see Figure 2). It consists of 134 lines of C code and generates a binary of length 18665 bytes, although we expect that the size can be reduced through compiler optimizations. The binder can execute multiple malwares and hides their displayed windows. It's default configuration parses it's own executable file for 256 byte long characteristic patterns, extracts the files between the patterns and executes the files. The malware has to be placed between patterns at an odd position proceeding the binder. The visible-started software is also placed between the patterns at an even position. If the content between two patterns is a malware, the binder hides its execution through standard Windows API calls. This may result in the following concatenation of binaries and patterns: (`binder` ‖ `pattern` ‖ `hidden malware` ‖ `pattern` ‖ `original software` ‖ `pattern` ‖ `hidden malware`). Because the operating system does not verify the structure of the infected executable file, the binder is able to execute the malware in parallel to the original application.

Efficiency. The `cyanid` proxy can infect binary executables up to 3 megabytes. While downloading an application of 3 megabytes a buffering delay of about 3 seconds is noticeable. The `calcium` binder works on Windows XP SP2 and Vista 32-bit Business. Depending on the malware used, the operating system shows warning messages produced by a bindshell for example. With a custom malware we were able to suppress the antivirus warnings. A test using an installer, which contained an embedded signature in its executable file, was successful on Windows XP and showed no warning messages. Windows Vista warned that the infected installed contained no signature.

[8] Tools like Tor provide a tunnel, which bypasses local and state-specific network nodes and provides anonymity for the client half of the network link, and there is no other (outband) information about him available.

6 Discussion

The current `cyanid` toolchain buffers the complete file, modifies it and then streams it to the users. Depending on the bandwidth and file size a delay is noticeable. Because the `calcium` infection routine only prepends the binder to the original file, a better proxy implementation would reduce the lag, which is currently generated by the buffering of the original file inside Privoxy.

The infection implementation — `calcium` — uses the binder technique to prepend malware ahead of the binary executable file. Other methods [16][24] include header infection with `AddressOfEntryPoint` modification, code section prepending, and cavity code, which have all in common that they modify the original file. Although the detection of the binder is very easy for an antivirus engine based on its binary sequence signature, methods exist to evade the antivirus detection, e.g., by encrypting the binder and thus the signature of its code.

A drawback of the current implementation is that the application icon, which is showed by the file browser, is changed to the application icon of the binder. This might raise suspicion by the user. However, the binder may modify its icon on each infection and mimic the application icon by simply copying the image of the icon in the original application file.

In our future work the `cyanid` implementation will get more targets like Mac OS X `dmg` containers and tarballs of source code. The `win32 PE` malware routine and the infection function might be enhanced to leave the icon and the file size unmodified.

References

[1] Linux Netfilter. `http://www.netfilter.org`.

[2] OpenVPN. `http://www.openvpn.org`.

[3] Privoxy. `http://www.privoxy.org`.

[4] Tor. `http://tor.eff.org`.

[5] Transproxy. `http://transproxy.sf.net`.

[6] Advanced Micro Devices, Inc. *AMD64 Virtualization Codenamed "Pacifica" Technology*, 33047-rev. 3.01 edition, May.

[7] A. Alkassar, M. Scheibel, C. Stüble, A.-R. Sadeghi, and M. Winandy. Security architecture for device encryption and VPN. In *Proceedings of Information Security Solutions Europe (ISSE 2006)*, pages 54–63. Vieweg-Verlag, 2006.

[8] Barnaby. Exploiting Embedded Systems. In *Blackhat Amsterdam*, 2006.

[9] Barnaby. Exploiting Embedded Systems The Sequel. In *CanSecWest*, 2007.

[10] Biondi and Ebalard. Fun with IPv6 routing headers. In *CanSecWest*, 2007.

[11] I. Corporation. LaGrande technology architectural overview. Technical Report 252491-001, Intel Corporation, Sept. 2003.

[12] W. Cui, R. H. Kat, and W. tian Tan. Design and implementation of an extrusion-based break-in detector for personal computers. In *Proceedings of the ACSAC*, 2005.

[13] S. Gajek, A.-R. Sadeghi, C. Stüble, and M. Winandy. Compartmented Security for Browsers – Or How to Thwart a Phisher with Trusted Computing. In *ARES 2007: Proceedings of the Second International Conference on Availability, Reliability and Security*. IEEE, 2007.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 193–206. ACM, 2003.

[15] Gill. ICMP redirects are bad. In *CYMRU*, 2002.

[16] Goppit. Portable Executable File Format - A Reverse Engineer View. *Code Breakers Journal*, 2(2), 2005.

[17] Greene. BGPv4 Security Risk Assessment, 2002. `http://www.cymru.com/Documents/barry2.pdf`.

[18] IETF. RFC 2818 HTTP Over TLS, 2000.

[19] P. Iglio and F. U. Bordoni. Trustedbox: a kernel-level integrity checker. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*. IEEE Computer Society, 1999.

[20] ISS X-Force. Mozilla SSL certificate spoofing (`http://xforce.iss.net/xforce/xfdb/16796`), Multiple vendor SSL intermediate CA-signed certificate spoofing (`http://xforce.iss.net/xforce/xfdb/9776`).

[21] G. Mohay and J. Zellers. Kernel and shell based application integrity assurance. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC '97)*. IEEE Computer Society, 1997.

[22] Ornaghi and Valleri. Man in the middle attacks demos. In *Blackhat US*, 2003.

[23] Pearce. Viral Polymorphism. 2003.

[24] Rozinov. PE File Infection Techniques, 2005.

[25] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research Division, Feb. 2005.

[26] A. Singh. Understanding Apple's Binary Protection in Mac OS X, 2006. `http://osxbook.com/book/bonus/chapter7/binaryprotection`.

[27] Skoudis. Don't discount software distribution sites as attack vectors, 2005. `http://SearchSecurity.com`.

[28] A. Spalka, A. B. Cremers, and H. Langweg. Protecting the creation of digital signatures with trusted computing platform technology against attacks by trojan horse programs. In *Sec '01: Proceedings of the 16th International Conference on Information Security: Trusted Information*, pages 403–419. Kluwer, 2001.

[29] Spangler. Packet Sniffing on Layer 2 Switched Local Area Networks. In *Packetwatch Research*, 2003.

[30] Steinhof, Wiesmaier, and Araujo. The State of the Art in DNS Spoofing. In *ACNS*, 2006.

[31] H. Thimbleby, S. Anderson, and P. Cairns. A framework for modelling Trojans and computer virus infection. *Computer Journal*, 41(7):444–458, 1999.

[32] van Hauser. Attacking the IPv6 Protocol Suite. In *Proceedings of the 22C3*, 2005. `http://www.thc.org/thc-ipv6/`.

[33] Whalen. An Introduction to ARP Spoofing. *2600 Magazine*, Fall 2001.

[34] D. Zovi and Macaulay. Attacking automatic wireless network selection. In *IEEE SMC IAW*, 2005.