Foreign Code Detection on the Windows/X86 Platform

Susanta Nanda Wei Li Lap-Chung Lam Tzi-cker Chiueh
{susanta,weili,lclam,chiueh}@cs.sunysb.edu
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

Abstract

As new attacks against Windows-based machines emerge almost on a daily basis, there is an increasing need to "lock down" individual users' desktop machines in corporate computing environments. One particular way to lock down a user computer is to guarantee that only authorized binary programs are allowed to run on that computer. A major advantage of this approach is that binaries downloaded without the user's knowledge, such as spyware, adware, or code entering through buffer overflow attacks, can never run on computers that are locked down this way. This paper presents the design, implementation and evaluation of FOOD, a foreign code detection system specifically for the Windows/X86 platform, where foreign code is defined as any binary programs that do not go through an authorized installation procedure. FOOD verifies the legitimacy of binary images involved in process creation and library loading to ensure that only authorized binaries are used in these operations. In addition, FOOD checks the target address of every indirect branch instruction in Windows binaries to prevent illegitimate control transfers to either dynamically injected mobile code or pre-existing library functions that are potentially damaging. Combined together, these techniques strictly prevent the execution of any foreign code. Experiments with a fully working FOOD prototype show that it can indeed stop all spyware and buffer overflow attacks we tested, and its worst-case run-time performance overhead associated with foreign code detection is less than 35%.

1. Introduction

During their day-to-day interactions with the Internet, users of Windows-based computers tend to knowingly or unknowingly download executable programs into their machines. For example, a remote binary program could be explicitly copied into the local disk as an FTPed file, an email attachment, a downloaded web object, etc., or implicitly downloaded as a plug-in or DLL when users click on a URL link. These foreign programs can be in the form of self-installing executables [1], ActiveX controls [2], scripts [6], or executable e-mail attachments such as WORD documents containing Visual Basic macros. With so many "entry points" through which remote malicious programs can enter a Windows PC, it has become very difficult to completely shut down all these entry points without affecting user productivity. Even for authorized applications, they may be vulnerable to control-hijacking attack [22, 28, 27], which could either execute instructions injected at run time or modify the control flow of the original program. We use the term *foreign code* to refer to binary executables, dynamically linked libraries (DLL), and instruction fragments that are loaded or injected at run time without proper authorization. Non-binary scripts are outside the scope of this paper.

Foreign code could range from benign programs that provide useful service to malicious ones that can compromise a user machine and turn it into a zombie host. A particular type of foreign code, *spyware*, has emerged as a serious threat to enterprise security, because they can cause various degrees of "soft damage" (listed in the order from low-risk to high-risk): unwanted interruption (pop-up ads), privacy violation (logging keystrokes and collecting web browsing patterns), damaging modification to system configurations or program settings, resource stealing (auto-dialing a toll-free number), and identity hiding (using a compromised host as a spam source or DDoS attack source). For example, BonziBuddy [37] is a spyware that is targeted at children, and interacts with the user as an agent that can talk, walk, browse, search, e-mail, download, etc., and while doing so monitors the surfing pattern and delivers ads based on search strings used. A typical spyware installs itself in such a way that it is automatically started after the host boots up. Other types of malware, e.g., virus and worm, can cause even more serious damage, but use a similar infection mechanism as spyware to compromise the victim machine.

Existing techniques to detect or stop foreign code are

limited because they only work on non-Windows platforms [15], or they focus on a specific type of attacks such as buffer overflow attack [32], or they could be easily defeated, e.g. shifting a DLL's base address by a few bytes to stop return-to-libc attacks [36]. None of them are able to provide a comprehensive defense against foreign code for commercially distributed binaries on the Windows/x86 platform. In this paper, we present a foreign code detection system called FOOD that could detect foreign code and prevent its execution. In particular, FOOD could stop spyware/malware by enforcing the constraint that the only binary programs that are allowed to run on a computer are those which are set up through an authorized installation process.

FOOD consists of two components as shown in Figure 1. The first component, the *Binary Image Verifier*, checks at process creation time and at library loading time to ensure that any executable file to be started or any DLL to be loaded must belong to a predefined list of authorized binaries, i.e., no dynamically downloaded (intentionally or not) executable files or DLLs could be started. The second component of FOOD, the *Target Address Monitor*, prevents illegitimate control transfer to injected or existing code during the execution of authorized applications by checking the target address of indirect control transfer instructions (indirect call/jump, and return). Moreover, unlike previous systems [38, 11], both components of FOOD are built on a binary analysis and instrumentation infrastructure called BIRD [26] and thus are capable of operating on Windows/X86 application binaries directly without requiring access to their source code or any other additional information such as debug/relocation symbols.

Currently, more and more enterprises are adopting a centralized application deployment architecture, in which a central server is responsible for installing and upgrading application binaries on the individual users' machines. In addition, the local disks of these desktop machines are periodically scanned to detect and remove malware. With the rampant threat of spyware, adware, viruses, and worms, some enterprises (for example, Computer Associates) are beginning to institute a policy that prohibits employees from downloading, installing, or running arbitrary executable binaries on their machines. This policy, if properly enforced, completely locks down the enterprise machines and leads to a client-server like computing environment that is as secure as a thin-client computing environment, where applications run on a central server rather than on individual desktops. FOOD is among the first known systems that can enforce this centralized binary control policy equipped with mechanisms to protect users from binary programs that are downloaded without their knowledge, as well as from malicious code that are injected into authorized applications.

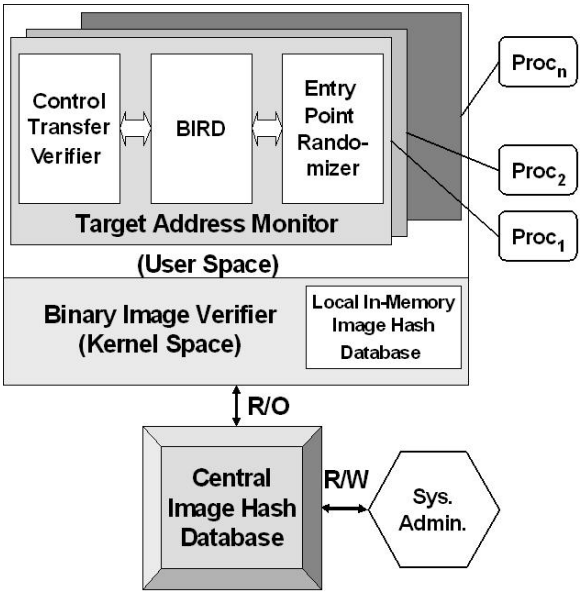

Figure 1: FOOD Architecture: FOOD consists of a Target Address Monitor (built on top of BIRD system) and a Binary Image Verifier. The former detects and stops control-hijacking attacks whereas the latter prevents execution of unauthorized binaries.

2. Related Work

Researchers have tried to solve the problem addressed by FOOD earlier, albeit in parts. The focus has been on detecting/preventing code-injection attacks [24, 14, 25, 18, 36], advanced attacks that rely on control-hijacking [11], or attacks launched through malicious external files [8, 34, 41, 9]. Besides sealing up the holes in many of these earlier approaches, FOOD also puts them together to build a more secured system.

Numerous techniques have been put up to thwart code injection attempts successfully. For instance, [24, 14] borrow ideas from the *human immune system* [10] and try to detect code injections by building mechanisms that can differentiate injected code from a process' own. One such mechanism scrambles a process' own code to produce some random bytes and descramble them just prior to execution. An injected code sequence, consisting of usually correct instructions, then stands out based on the content. However, apart from the apparent performance penalty associated with such scrambling/descrambling (reported slowdown is a factor of 3 to 5), they are also susceptible to attacks that do not inject code, e.g. the ones that disrupt control flow by overwriting critical data structures. On the other hand, FOOD tries to distinguish a process' own code from foreign code based on the location. By forcing the injected code to land up either in heap or stack, FOOD ensures foreign code always differ in location from a process' own code, which lives in code sections. A similar approach is used to detect return-to-libc kind of attacks too. Because

most instructions execute natively on the processor, the performance overhead of FOOD is reasonable.

Determina's SecureCore [18] is another solution to prevent code-injection attacks that borrows ideas from [12, 13, 16, 25]. It is built on top of a dynamic interpreter and enforces security checks while a program is being interpreted. The checks could include restriction of privileges based on code origin, restricting control transfers, and ensuring that the user-specified sandboxing checks will never be bypassed. SecureCore leverages these technologies to counter memory-based attacks, e.g. buffer overrun. Microsoft's next-generation secure computing base [8] proposes the idea of isolating a process and its data from the rest of the system to safeguard it. However, it still does not eliminate attacks like return-to-libc.

PaX [36] is a Linux kernel patch that uses segmentation hardware to prevent execution of injected code. By refusing code execution on writable pages and modifying `mmap()` to randomize the library base address, it also makes it more difficult to mount return-to-libc attacks. However, this idea does not work for the Windows OS, because Windows applications expect some of its core system libraries to be mapped to constant locations in the virtual address space. Openwall [31], Exec-Shield [23], and W$\wedge$X [35] are a few similar tools for specific Unix-like systems.

Control flow integrity (CFI) enforcement [11] is shown to be an effective approach towards protecting an application from control-hijacking attacks as almost all of them try to change the control flow in some way or the other. [11] works on top of vulcan [38], a binary analysis and instrumentation tool, to get around the Windows disassembly problems. Vulcan requires comprehensive debugging information for the associated binaries to be able to analyze them. However, third party applications are unlikely to be distributed with their symbol and relocation information, which is why FOOD chooses not to take this route.

In addition to preventing control-hijacking attacks, FOOD is also designed to prevent attacks launched through external malicious executables, such as EXEs, DLLs, ActiveX objects, etc. A number of techniques have been put up by researchers to address this issue at various levels. For example, [3, 4, 39] associate additional information like hashes and digital signatures/certificates with files to verify their integrity, authenticity, and non-repudiability. Problems addressed by this kind are orthogonal to FOOD and are leveraged to identify potentially malicious files. Likewise, [9] can be leveraged too, which uses signature look up to filter out unwanted software.

Building defense mechanisms within the end-hosts has also been identified as an alternative. For instance, SecureWave's sanctuary [34] product stores and uses hashes of binary files to prevent unauthorized programs from executing. However, it remains susceptible to control-hijacking attacks. The IMP tool [41] is another related tool that tracks files entering through web browsers, Windows explorer, and e-mail clients, and prevent them from doing bad things by monitoring them. Unlike FOOD, it does little about tracking DLLs, and files entering through command-line tools. Finally, Windows Vista [7] seems to propose a wide range of improvements to detect foreign code; however, its too early to predict its effectiveness.

3. Binary Image Verification

When a software package is officially installed, e.g. by an administrator, FOOD computes MD5 hash values of the package's binary files and stores the hash values, together with their full path name, file size, creation time, etc., in an *image hash database*. When a binary image is to be loaded into a process' address space, either at the process start-up time or during its run time, FOOD computes the image's MD5 value, checks it against the image hash database, and continues with the load operation only when there is a match in the image hash database. If the computed hash value does not match any entry, the corresponding operation is aborted, and a report entry is added to an event log.

The idea of checking a binary image's hash value before executing it is not new, and is heavily used in verifying the origins of binaries downloaded from the Internet. The image hash value forms the digital signature or digital certificate [39] of these binaries. Similarly, file image hash has also been the basis for checking the integrity of important files in Tripwire [40]. FOOD takes this idea one step further and applies it to the binary image used in process creation and dynamic library loading. As a result, once FOOD is deployed on a computer, only "officially installed" binaries can execute on that computer.

The *binary image verifier* is implemented as a kernel driver that registers a callback function with the Windows kernel using the API, *PsSetLoadImageNotifyRoutine()* [5]. This callback function is invoked whenever a binary image is mapped into the user/kernel portion of a process' virtual address space, and before the execution of the image begins. The system call, *ZwMapViewOfSection()*, which maps binary images and triggers these callback functions, will not return until each of these callback functions completes. Using this mechanism, FOOD inserts code to verify the binary image used in every library load, process creation, and OLE/ActiveX control load.

Windows environment also offers multiple technologies for component/code reuse that include OLE, COM, and ActiveX. ActiveX, in particular, is very popular and is typically used to display active contents in web pages. At run time, when the container application (e.g. a web browser) displays a page that has active content embedded, the corresponding object's executable, DLL or OCX (OLE Com-

ponent Extension), is downloaded and stored in the local disk. Thereafter, for the current as well as each future use of the same object, the binary is loaded and executed either in the browser's address space (ActiveX DLL/OCX), or as a standalone process (ActiveX EXE). FOOD disallows the execution of all ActiveX objects that are not officially installed on the machine, except ActiveX scripts, which are executed by a scripting engine.

One way to bypass FOOD's binary image verification process is to unload the corresponding kernel driver. However, FOOD disallows unloading of the image verification driver at run time. Because the image database holds hash values for all the installed binaries, binary image verification will be voided if the database or its communication with the kernel driver is compromised. To eliminate these vulnerabilities, the image hash database, managed centrally by the system administrator, is sent to each individual machine and loaded into the FOOD driver's memory as part of its boot-up and initialization process. The integrity of the image hash database during initialization is assured through encryption based on the system administrator's private key. The public key is installed in each of the machines as a part of FOOD's deployment and is cross-checked with the central server before being copied to the FOOD driver's memory at boot up time. After new binaries are installed, the central server sends an updated version of the image hash database to user machines using the same encryption method.

FOOD's binary image verification approach can effectively eliminate most if not all known spyware/adware and worm attacks. However, this effectiveness comes at the expense of convenience: A machine's user loses the flexibility of installing additional software packages. In an enterprise environment, the additional security protection brought forth by this approach is well worth the inconvenience (though folks as jericho forum [20] might have a conflict here), but may not be so in other operating environments.

4. Target Address Monitoring

4.1. Overview

Control-hijacking attacks modify certain control-sensitive data structure in the victim application, such as return address, function pointer and import/export table entry, so as to steer its control to dynamically injected code or existing code. One way to stop control-hijacking attacks is to check the target address of every control transfer instruction at run time and make sure control is always transferred to valid places in the address space. Because direct branch instructions, by definition, hard-code their target addresses in the instructions themselves and the code sections are read-only, it is not possible to tamper with the target addresses of such instructions at run time. Therefore, only the target addresses of indirect branch instructions, i.e., indirect jumps, indirect calls, and return instructions need to be monitored and checked.

The exact criterion used in determining whether an indirect branch's target address is *valid* or not is different for different types of control-hijacking attacks. For code-injection attack, which injects a piece of code into the victim application's address space at run time and steers the application's control to a piece of injected code, a target address is valid if and only if it belongs to the code section. Here we assume the code section is read-only; therefore, injected code must reside in the heap or the stack. However, treating any instructions outside the code section as an attack code could lead to false positives because there are legitimate reasons to put instructions on the stack or heap. For example, Linux's signal delivery mechanism puts a trap instruction on the user stack so as to transfer control to the kernel after a signal handler is finished. As another example, the trampoline construct that GCC supports requires putting instructions on the stack to implement nested function calls. Fortunately, in the Windows environment, neither is a problem. To the best of our knowledge, trampoline is not supported by any Windows compilers [33], including Visual Studio, and Windows does not need to put traps on the user stack as it uses well-defined library routines for signal handling [21] that traps back into the kernel once handler routines have finished execution (refer [26] for details).

A return-to-libc attack does not inject any code but attempts to steer the victim program's control to the entry point of some shared library function. By supplying appropriate arguments to these library routines, the attacker inflicts the desired damage. To stop such an attack, FOOD redirects all the calls to shared library functions to a new set of entry points whose locations are randomly chosen at run time to prevent guessing. FOOD detects a return-to-libc attack by checking if the target addresses of indirect control transfer instructions fall outside of the code section and the randomized entry point region. Currently, we only redirect calls to common shared libraries such as `KERNEL32.DLL`, `NTDLL.DLL`, and `USER32.DLL` that are more likely to be targets of return-to-libc attacks. Since attackers do not know the library functions' new entry points, they will continue to use their old entry points and eventually be detected by FOOD. Because Windows expects native functions to be present at fixed addresses, DLLs cannot be moved around arbitrarily in the address space, and `NTDLL.DLL` is one such example DLL. For this reason, FOOD decides not to play with their base addresses and concentrates one level upstream to solve the problem. Linux, having no such difficulty, could use the technique as discussed in [36, 15]. Section 4.3.3 details the implementation scheme.

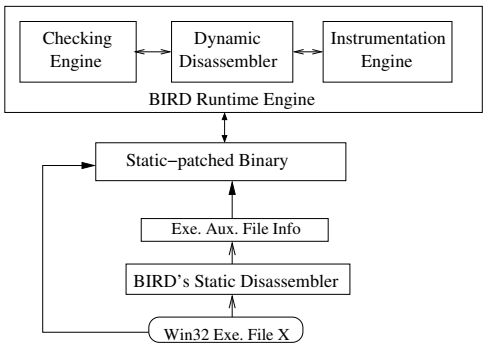

Figure 2: BIRD's architecture consists of a static disassembler and a run-time engine, which in turn consists of an instrumentation engine, a checking engine, and a dynamic disassembler. All the indirect branches of an application are patched so that BIRD's checking engine takes control at these branches.

FOOD's target address monitor is designed to work on arbitrary Windows/X86 binaries. Ideally, the target address checks should be inserted into the binary files statically. However, because it is impossible to completely disassemble all commercially distributed Windows/X86 binaries statically, FOOD's target address monitor is built on a binary interpretation and instrumentation engine called BIRD [26]. The rest of this section will describe BIRD and the extensions we added to prevent control-hijacking attacks.

4.2. The BIRD System

4.2.1. Big Picture. BIRD is designed to be an infrastructure that facilitates analysis and instrumentation of Windows/X86 binaries during the development of software security systems. Figure 2 shows the architecture of BIRD. It employs a static-cum-dynamic disassembly approach to achieve 100% executed code coverage for arbitrary Win32 binaries. Given an input binary, BIRD disassembles it statically as much as possible, without compromising its accuracy. Due to space limitations, details of the disassembly algorithm are not covered here; interested readers can look them up in [26]. After the static disassembly pass, the input binary is divided into two disjoint sets of code areas. Those that are known to contain instructions are called *known areas* (KA) and the others are called *unknown areas* (UA), which will be disassembled at run time. Because the static disassembler has fully explored the targets of all direct branch instructions, the only way a program's control can go from a KA to an UA is through an indirect branch. BIRD's static disassembly engine produces two auxiliary files to be used by its runtime engine: one for the address ranges for UAs/KAs, and the other containing information for indirect branch instructions.

4.2.2. Interpretation. In order to guarantee that every instruction in a binary file is "examined" before its execution, BIRD's runtime engine takes control at each indirect branch. This is achieved by replacing each indirect branch instruction (done statically after the disassembly phase) with the instruction, `call check()`, such that the control is redirected to a routine in BIRD's runtime engine before being transferred to its target address. Each time when BIRD's runtime engine takes control, it identifies the associated indirect call/jump, calculates its target address, and checks if the address falls in an UA or a KA. If the target falls in a KA, BIRD lets it go through; otherwise a dynamic disassembler is invoked to disassemble the corresponding UA as much as it can. BIRD's runtime disassembler uses a simpler disassembly algorithm that linearly sweeps an UA starting with the branch target until it encounters an indirect branch instruction.

BIRD does not intercept *return* instructions as there are few real world Windows applications that modify a return address after it is pushed on to the stack. Kernel callbacks, such as exception handlers, callbacks, and asynchronous procedure calls (APCs), are invoked through function pointers, i.e. indirect calls, from user-level library routines in `NTDLL.DLL` and `USER32.DLL`. The fact that BIRD can intercept all indirect calls in these libraries implies it can intercept all kernel callbacks as well.

4.2.3. Instrumentation. In addition to interpretation, BIRD also provides an instrumentation service: Given an instrumentation point in the form of a virtual address and an instruction sequence, BIRD logically inserts the instruction sequence at the specified point and resumes execution from that point onwards. In fact, BIRD itself uses binary instrumentation to redirect all indirect calls/jumps to its runtime engine. Binary instrumentation introduces the following two problems: (1) Find enough space to hold the instrumentation code, and (2) Replace instrumentation point by a branch instruction so as to redirect control to the instrumentation code.

BIRD solves (1) by statically allocating space in a new code section in the binary file and putting all the instrumentation code therein. Any new instrumentations required at run time use this space to place instructions. (2) is trivial whenever there are enough bytes at the instrumentation point to hold the branch instruction (5 bytes long), which is true in most cases. However, when the instruction at an instrumentation point is shorter than 5 bytes, BIRD replaces the instruction and its follow-on instructions with the branch instruction, `call instrument_code()`, if none of the replaced instructions is a target of any direct branch in the same function. If such a replacement is not possible, the one-byte breakpoint instruction (`0xCC` on X86) is used instead. By installing an exception handler for the breakpoint exception, BIRD's runtime engine gets control at breakpoints. Figure 3 illustrates an example in-

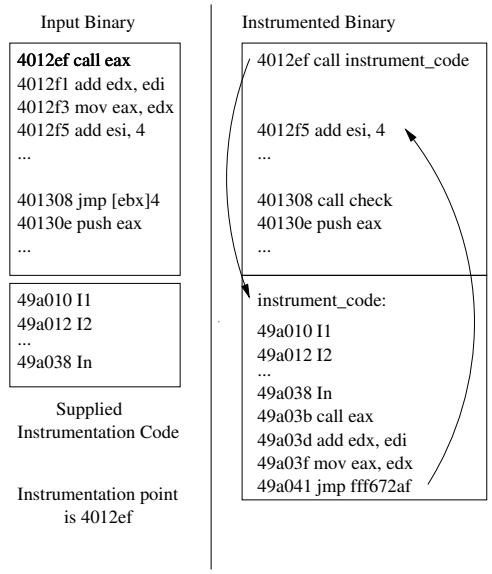

Figure 3: BIRD instruments at a program point by replacing the instructions by a call to a routine injected into the application, *instrument_code()*, and storing the overwritten instructions in some safe place. At run-time when *instrument_code()* gets invoked, it uses them and manages a transparent execution of the replaced instructions and any instrumented code, if at all.

strumentation. For more details, readers are advised to refer to [26].

4.3. From BIRD to FOOD

4.3.1. Preventing Unauthorized Writes. Because control hijacking attacks modify control-sensitive data structures, FOOD thwarts these attacks by turning as many data areas into *read-only* as possible. A Windows PE binary contains a number of sections relevant to data that include *.idata* (import table), *.edata* (export table), *.data* (initialized data), *.bss* (uninitialized data), *.crt* (initialized data for C/C++ runtime libraries), *.tls* (thread local storage), *.reloc* (relocation information), *.rsrc* (resources), and one or more user-defined data sections (using `#pragma data_seg` directive).

Every section in a PE file, except *.reloc*, is loaded into a process' virtual address space and more often as *writable* if they contain data. Because *export table* never needs to be modified, FOOD first marks *.edata* as *read-only* soon after it is loaded. The *.idata* section of a binary corresponds to its import address table (IAT) [17], which contains information regarding the functions the binary imports from other modules. IAT needs to be filled in with the real addresses of the functions by the Windows loader [30], and is updated just *once* for an application. Thus, although *.idata* is marked read-write at the start, FOOD is able to mark it *read-only* after it has been updated by the loader without breaking the application. Other sections contain data that are frequently modified and can not be made read-only.

4.3.2. Preventing Code-Injection Attacks. To prevent unauthorized control transfers, FOOD monitors the target addresses of all indirect branches, i.e. indirect calls, indirect jumps, and returns. Because BIRD already monitors targets for indirect jumps and calls, FOOD only intercepts `return` instructions in addition, by leveraging the instrumentation services provided by BIRD. If the target address of an indirect branch falls outside the code region, which includes the DLL regions and is generally marked as read-only, FOOD raises a code-injection attack alert. Although modern X86 hardware starts to support XD (executable disable) bit, which could be used to stop code-injection attacks, FOOD's code-injection attack prevention mechanism can protect legacy platforms that do not support XD bit.

On the X86 architecture, a return instruction is either 1 byte (`ret`) or 3 bytes (`ret n`) long. Because the `call check()` instruction takes 5 bytes, instrumenting return instructions is as complicated as instrumenting short indirect jumps/calls, which are 2 bytes long. Since a `return` instruction is the last instruction in a function, FOOD tries to replace it along with the preceding instructions by `call check()`. In most cases, returns are preceded by instructions that adjust the stack pointer and reload the old frame pointer (`ebp`). These instructions are rarely the target of any direct branch and thus are safe replacement candidates. If enough bytes could not be found before a `return` instruction, FOOD falls back to the breakpoint instruction. Because the replaced instructions logically precede the `return` instruction, they are executed prior to the target address checks in BIRD's checking engine.

4.3.3. Preventing Return-to-LIBC Attacks. In a PE binary, all calls to library functions are *typically* funnelled through the IAT, whose entries contain pointers to imported functions. One can call an imported function using an indirect call instruction that uses the corresponding IAT entry as its target. However, in most PE binaries, library function calls are implemented as a call to a *thunk* that exists in the application's code region. The thunk for each imported library function contains an indirect jump that reads the IAT entry for the associated library function. In some sense, this thunk is effectively an application's entry point into the corresponding library function. An example is shown below.

```
*** .idata section start ***
...
77879426                [Addr:00040042]
(Entry to GetMessage() in User32.dll)
...
*** .idata section end   ***

*** .text section start  ***
JMP DWORD PTR(00040042) [Addr:00015794]
...
CALL 00015794  (call to GetMessage())
*** .text section end    ***
```

In general, a *return-to-libc* attack could modify the target address of an indirect branch instruction to point to the entry point of the target library function (`77879426`, in the above example), or the corresponding thunk (`00015794`), or any call site to this library function in the application (the address of `CALL 00015794`). To stop return-to-libc attacks, FOOD first scans an application's binary to identify the addresses of the thunks for all imported functions, and the addresses of the entry points for all non-library functions. Then FOOD checks if any of these addresses are used in any assignment operations. If an entry-point address indeed appears in an assignment operation, it means that the original application program intends to use it as a target of indirect calls. These addresses thus form the application's *legitimate function pointer set*. Here FOOD makes the assumption that a call instruction can only go to a non-library function's entry point or a library function's thunk.

Given an indirect call instruction, FOOD declares it to be an intrusion if its target address is not in the legitimate function pointer set. With this mechanism, the attacker cannot steer a victim program's control to a DLL region because it is not in the legitimate function pointer set. Similarly, the attacker cannot steer a victim program's control to a thunk address or a call site address associated with a library function if they are not in the legitimate function pointer set.

Given an indirect jump instruction, FOOD declares it to be an intrusion if its target is not within the same function it is in. Given a return instruction, FOOD declares it to be an intrusion if its target instruction is not preceded by a control transfer instruction that transfers control to the function containing the return instruction.

4.4. Self-Modifying Code

More and more Windows binaries (especially shareware) are transformed by binary encryption tools such as UPX [29] and distributed in an encrypted format, which typically consists of a plaintext decryption engine and an encrypted version of the original binary. At run time, the control first goes to the decryption engine to recover the plaintext version of the original binary and then jumps to the entry point of the original binary. It is possible to run a Windows binary through a binary encryption tool multiple times. In this case, the decryption engine for the original binary is itself encrypted and needs to be decrypted at run time. Because of run-time decryption, this type of binaries are self-modifying code, which makes both binary interpretation and foreign code detection more challenging.

To handle this particular type of self-modifying code, BIRD needs to mark pages in the known area as read-only. Whenever a read-only known page is modified, it is turned into read-write and unknown. As a result, when the program control subsequently is transferred to this unknown page, it needs to be re-disassembled. Moreover, because code is generated dynamically, BIRD needs to intercept both direct and indirect branches statically and in the beginning of execution, and only intercepts indirect branches after the program reaches the main entry point of the original binary, which lies in a well-known region for most Windows binaries. Here we assume that the original binary is not modified and it is loaded into the address space as if it is started directly.

Self-modifying code also poses problems for the original FOOD because its assumption that the attacker cannot modify the code pages no longer holds. However, for self-modifying code generated by UPX-like tools, this is not a problem. Because FOOD marks all pages in the code section as read-only only after the program's control is transferred to the main entry point of the original binary, any subsequent attempts to modify code pages, including those from the attackers, are intercepted and rejected.

5. Attack Analysis

Indirectly Calling a Library Function. If an application function calls a sensitive library function using the standard import table mechanism, the attacker can steer an indirect branch to the corresponding call instruction in this application function. FOOD can detect such attacks as long as the entry points of the "wrapper" functions are not in the legitimate function pointer set. The same conclusion applies to the cases in which the attacker can go upstream several steps by calling a function that calls a function that calls a function that calls a library function, etc.

In Windows, system DLLs such as `NTDLL.DLL` are mapped to fixed locations in a process' address space leading to its misuse as poor programmers sometimes hardcode the entry points. As an example, an instruction like `call 77879426` could be used to invoke `GetMessage()` from `USER32.DLL`. Since these are direct calls, FOOD cannot intercept them. If an attacker can steer an indirect branch to jump to such direct calls, FOOD cannot detect such attacks. Although we never experienced such a case, this attack cannot be ruled out. Direct jumps from one internal routine to another within a DLL pose a similar threat.

Modifying Control Flow without Changing Target Address. Even without modifying any of the control data structures of a program, it is possible for an attacker to steer the program control by modifying certain sensitive global variables. For example, modification of the variable like `UID`, or a security handle can do wonders. Moreover, such global variables are stored together with other data structures that need to be modified, and thus cannot be marked as read-only. FOOD can do little to prevent this type of attacks because it only checks the target address of indirect branches.

Kernel Vulnerability. Since FOOD does not track instruc-

Attack Type	Attack Scenarios
Spyware/Adware	- Out-process COM: e.g. ActiveX EXE - In-process COM: ActiveX DLL/OCX
Code-Injection	- Buffer overrun into stack - Shell-code injected into heap/stack - Format-string into stack/heap
Return to LIBC	- Modifying function pointers to point to Libc - Buffer overrun, modify return address to point to a library routine - Buffer overrun, modify return address to point to an IAT thunk - Modifying function pointer to point to an IAT thunk
Return to Return	- Modifying return address to point to `ret` in application - Modifying return address to point to `ret` in DLL
Import/Export Tables	- Modifying IAT entry to bypass FOOD's additional thunk - Modifying Export table entry

Table 1: List of different kinds of attacks that FOOD stops successfully. IAT thunk is an instruction that eventually jumps to a target library routine using an address stored in import address table.

tions executed in kernel mode, a smart attacker could exploit vulnerabilities in the kernel code to disable checks introduced by FOOD. However, this attack becomes less likely when no unauthorized code is allowed to execute in user space.

Scripts and Interpreted Code. If a piece of foreign code is written in a scripting or interpreted language, such as Java, Perl, or Visual Basic, FOOD cannot detect its execution, let alone stop it as it *only* tracks execution of binaries. Fortunately, it is relatively straightforward to introduce additional security checks into a script's interpreter. So comparatively, malicious attacks written in a scripting language are a lesser threat than those in the form of executable binaries.

6. Evaluation

6.1. Effectiveness

We applied a set of known attacks listed in Table 1 to evaluate the effectiveness of FOOD. As expected, FOOD was able to stop all of them. In addition, we applied the Blaster worm and the Slammer worm attack, both of which exploit buffer overflow vulnerability, and FOOD was able to thwart both attacks successfully. We also tried another real-world attack that exploits Windows GDI+ JPG vulnerability [19]. The exploit corrupts a memory location holding a global variable pointing to a C++ method. FOOD detected this exploit successfully when the function is invoked through the modified pointer.

Because real-world return to libc attacks, on the Windows platform are rare, we chose to hand-craft several synthetic return-to-libc attacks, and once again FOOD succeeded in detecting all of them.

Appl.	Native Exec.	No. of RETs	BIRD Ovhd	RET Ovhd	FOOD Ovhd
comp	1806.2M	1425K	5.6%	15.0%	20.6%
compact	1253.8M	372K	1.4%	5.8%	7.2%
find	14076.0M	11485K	8.5%	22.6%	31.1%
MD5	292.1M	25K	0.8%	2.5%	3.3%
lame	653.0M	292K	0.9%	10.8%	11.7%
sort	393.5M	251K	5.0%	12.0%	17.0%
ncftp	1725.5M	1520K	5.3%	17.0%	22.3%

Table 2: The FOOD prototype is built on top of BIRD and adds a check at every return instruction, incurring more overhead than BIRD. Native Exec shows the native execution time in no. of CPU cycles. RET Overhead, is the additional overhead over BIRD's overhead, as represented by a percentage with respect to the native execution time. Thus, the total overhead is the sum of BIRD's overhead and that spent in checks for return instructions.

6.2. Performance

Many large Windows applications, including MS Office suite, Internet Explorer, Acrobat reader, etc., have successfully run on the FOOD prototype. Several server applications also ran successfully as described in the previous subsection. However, it is difficult to isolate FOOD's run-time overhead using interactive applications. Instead, to simplify end-to-end delay measurement, we chose to use the following seven batch programs to study FOOD's performance overhead: *comp* (comparing two 4MB files), *compact* (compressing a set of ten executables in a directory), *find* (finding a given string from a 500KB DLL file), *MD5* (computing the hash value for a 5MB executable), *lame* (converting a wav audio file to MP3 format), *sort* (sorting a 500KB ASCII file), and *ncftp* (getting a 1.5M file through file transfer protocol from a remote machine). All the performance results reported in this subsection, are based on experiments that are carried out on a uni-processor P4/1.8GHz/512MB Windows XP machine.

FOOD's overhead arises from the target address monitoring mechanism at run time and the binary image verification at start-up time. The *target address monitor* overhead, in turn, is due to BIRD and the instrumentation at *return* instructions. Table 2 shows the run-time overhead of FOOD's *target address monitor* for seven Windows batch applications. As expected, the overhead is mainly determined by the dynamic count of indirect branches. GUI- and I/O-intensive applications are expected to incur less overhead compared to CPU-intensive applications, which tend to invoke a large number of functions and thus return instructions per unit time. Since CPU cycles are the only resources consumed by FOOD, CPU cycle overhead directly translates to the worst-case (as I/O may overlap with some of that) runtime overhead incurred by FOOD. Although the performance overhead of the current FOOD prototype is not negligible (less than 32%), it is much more efficient than instruction-set randomization techniques [24, 14] that incur 100% to 500% overhead.

Appl.	Native Execution	Check Overhead	Interpretation Overhead	BIRD Overhead
comp	1806.2M	5.6%	0.0%	5.6%
compact	1253.8M	1.2%	0.1%	1.4%
find	14076.0M	8.5%	0.0%	8.5%
MD5	292.1M	0.1%	0.3%	0.8%
lame	653.0M	0.5%	0.2%	0.9%
sort	393.5M	4.8%	0.1%	5.0%
ncftp	1725.5M	5.3%	0.0%	5.3%

Table 3: Performance overheads of the BIRD prototype for seven batch programs. The performance overhead arises from additional checks at indirect branches (Check Overhead), dynamic disassembly (Interpretation Overhead), and others e.g. breakpoint overheads (not shown in the table). Native execution time is reported in number of CPU cycles (M = 10^6).

Application	No. of RETs	RET Ovhd.	BIRD Ovhd.	FOOD* Ovhd.
comp	1425K	17.1%	5.6%	23.0%
compact	372K	6.9%	1.4%	9.7%
find	11485K	24.9%	8.5%	35.1%
MD5	25K	3.3%	0.8%	4.8%
lame	292K	12.1%	0.9%	14.8%
sort	251K	13.7%	5.0%	20.2%
ncftp	1520K	19.4%	5.3%	26.4%

Table 4: FOOD equipped with return-to-libc attack detection mechanism (hence FOOD*) implemented on top of BIRD. RET Ovhd shows the overhead due to checks at return only, which is incurred in addition to the overhead of BIRD (BIRD Ovhd.). The total overhead, shown as FOOD* Ovhd, also includes the overhead due to additional range checks that are added for the indirect branch targets in this phase.

Table 3 shows the break-down of the performance overhead of the seven batch applications running under BIRD. BIRD's overhead comes from several sources; *Check Overhead* comes from `check()` invocations at each indirect call/jump, *Interpretation Overhead* is due to dynamic disassembly, and some others e.g. breakpoint overhead (not shown). The overall run-time overhead of BIRD varies between 1% to 6% except *find*, which executes an inordinate number of indirect branches.

FOOD's *return-to-libc* attack detection introduces additional penalty, because it requires more checks to be performed at each of the indirect branches. The base version of FOOD's *target address monitor* only aimed to stop code-injection attacks and therefore only checked whether the indirect branch targets fell outside the code section. To prevent return-to-libc attacks requires an additional range check for the target address depending on if the instruction is a return, an indirect jump or an indirect call. Moreover, it also incurs an additional overhead to derive an application's legitimate function pointer set. Table 4 shows the results after this mechanism is incorporated into FOOD, shown as FOOD*.

Besides monitoring control targets, FOOD also verifies a binary image when it is mapped into an application process' address space. For each such image, FOOD computes its MD5 hash and checks it with the image hash database (see section 4). Table 5 shows the overhead incurred in this process. Although the absolute overheads appear substantial, they are all less than 1 second and thus have minimal effects on GUI-based or CPU intensive applications. Moreover, almost all binary images in these tests (except those loaded through `LoadLibrary()` calls) are loaded during the application start-up time, and the resulting overhead is nearly constant and independent of the application execution time.

Appl.	No. of Images	Image Verif. Ovhd. in cycles	Overhead in Absolute Time
comp	25	1729M	0.96s
compact	24	1707M	0.95s
find	25	1723M	0.96s
MD5	3	152M	0.08s
lame	3	179M	0.10s
sort	24	1701M	0.94s
ncftp	19	477M	0.27s

Table 5: FOOD's Image Verification Overhead shown, both in terms of number of CPU cycles (third column) and absolute time (fourth column) on a 1.8GHz Pentium IV CPU. The second column shows the number of binary images, EXEs and DLLs, mapped into the process' virtual address space during the application's execution.

Most of the libraries that are imported and mapped into the application's address space fall into a common group that are used by most other applications. Since DLLs are supposed to be shared, each DLL, once loaded into the memory, only needs to be checked once. Therefore the image verification cost of a DLL can be amortized over its entire life time, which could be days for popular DLLs. One possible way to further reduce the binary image verification overhead is to integrate image verification with page fault handling so that a page fault handler can perform image verification whenever it brings a page from disk into physical memory.

7. Conclusion

An enterprise computing environment that is built on the *thin client* architecture is generally believed to be more secure than that based on the *client server* architecture, because it is easier to centrally manage and control the set of applications that are allowed to run on the central terminal server. The foreign code detection system described in this paper, FOOD, attempts to approximate the security model of the thin client architecture on a client server architecture without suffering from the former's network bandwidth and latency problem. Moreover, FOOD can effectively thwart most spyware and adware, because most if not all of them involve binaries that are downloaded without being pre-authorized and thus will be rejected at execution time. Finally, FOOD offers more than whole-binary integrity check because it could further prevent corruption of authorized applications by checking at run time the target address of every indirect branch in Windows binaries, including dynamically linked libraries.

Combining these techniques together, FOOD represents one of the most comprehensive foreign code detection system on the Windows platform that includes protection against code-injection attacks, return-to-libc attacks and malicious DLL or plug-in downloads. Performance measurements of the first FOOD prototype show that the start-up latency penalty associated with binary image verification is less than 1 sec for large Windows applications such as those in MS Office suite, and the execution time overhead for target address monitoring is less than 35%, which is not unreasonable when we look at the level of protection it offers.

References

[1] About windows installer. http://msdn.microsoft.com/library/en-us/msi/setup/about_windows_installer.asp.

[2] Introduction to activex controls. http://msdn.microsoft.com/library/default.asp?url=/workshop/components/activex/intro.asp.

[3] Introduction to code signing. http://msdn.microsoft.com/library/default.asp?url=/workshop/security/ authcode/intro_authenticode.asp.

[4] Introduction to code signing. http://msdn.microsoft.com/library/default.asp?url=/workshop/security/authcode/signing.asp.

[5] Microsoft msdn library. http://msdn.microsoft.com/library/.

[6] Windows script host. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ script56/html/ wsoriWindowsScriptHost.asp.

[7] Windows vista security and protection. http://www. microsoft.com/technet/windowsvista/security/default.mspx.

[8] Microsoft next-generation secure computing base. http:// www.microsoft.com/technet/archive/security/news/ngscb.mspx, July 2003.

[9] Windows antispyware (beta): Analysis approach and categories. http://www.microsoft.com/athome/security/ spyware/software/isv/analysis.mspx, March 2005.

[10] S. H. A. Somayaji and S. Forrest. Principles of a computer immune system. In *New Security Paradigms Workshop, ACM*, pages 75–82, 1997.

[11] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, Nov 2005.

[12] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[13] I. Baron. Dynamic optimization of interpreters using dynamorio. S.M. Thesis, MIT, September 2003.

[14] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[15] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, August 2005.

[16] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

[17] P. Dabak, M. Borate, and S. Phadke. Undocumented Windows NT. M and T Books, October 1999.

[18] Determina. Introducing memory firewall technology. http://www.determina.com/docs/Determina2004.

[19] E. Florio. Gdiplus vuln - ms04-028 - crash test jpeg. full-disclosure at lists.netsys.com, Sept 2004.

[20] J. Forum. Visioning white paper. http://www.opengroup.org/projects/jericho/uploads/40/6809/vision_wp.pdf, February 2005.

[21] Gamedev. Windows seh and c++ exception handling. http://www.gamedev.net/reference/articles/article1291.asp, January 2001.

[22] Imperva. Buffer overflow attack. http://www.imperva.com/application_defense_center/glossary/buffer_overflow.html.

[23] R. Inc. New security enhancements in red hat enterprise linux. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf, August 2004.

[24] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM Press, 2003.

[25] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, 2002.

[26] S. Nanda, W. Li, L.-C. Lam, and T. cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *The 4th Annual International Symposium on Code Generation and Optimization*, 2006.

[27] Nergal. The advanced return-to-lib(c) exploits. *Phrack*, 58(4), December 2001.

[28] T. Newsham. Format strings attacks. http://www.securityfocus.com/guest/3342, September 2000.

[29] M. F. Oberhumer and L. Molnr. Upx: the ultimate packer for executables. http://upx.sourceforge.net/, June 2004.

[30] R. Osterlund. What goes on inside windows 2000: Solving the mysteries of the loader, March 2002.

[31] O. P. Page. Information security software for open environments. http://www.openwall.com/.

[32] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conference*, pages 211–224, 2003.

[33] D. B. Probert. Windows kernel internals. http://www.i.u-tokyo.ac.jp/ss/lecture/new-documents/ Lectures/05-TrapsInterruptsExceptions/ TrapsInterruptsExceptions.pdf.

[34] SecureWave. Securewave sanctuary. http://www.securewave.com/Sanctuary_AC.jsp.

[35] SecurityDocs. Openbsd security. http://www.securitydocs.com/library/3100.

[36] B. Spengler. Pax: The guaranteed end of arbitrary code execution. http://www.grsecurity.net/PaX-presentation.ppt.

[37] SpywareGuide. Bonzibuddy. http://www.spywareguide.com/product_show.php?id=512.

[38] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

[39] TechTarget. Digital certificate. http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci211947,00.html, 2001.

[40] Tripwire. http://www.tripwire.com/.

[41] J. A. Whittaker and A. D. Vivanco. Neutralizing windows-based malicious mobile code. In *SAC 2002*, March 2002.