CFI on Malware

Arun John Kuruvilla
New York University

Dushyanth NP Chowdary
New York University

Jonathan L Poch
New York University

Mohammed Ashraf Ali
New York University

Abstract

A common property most modern exploits leverage is the manipulation of control flow. In this paper, we present plugins for PANDA that enforce a system-wide control flow integrity policy, using a combination of whitelist and shadow call stack based approaches that monitor control flow transfers and detect violations. Replays of a simple stack exploits are created using PANDA and analyzed to identify when control flow integrity is broken. We test our plugins on benign replays as well as the replays with successful stack exploitations.

1 Introduction

Vulnerable software has been a major factor in security breaches around the world. Flaws in logic, negligent mistakes, and misunderstandings have opened doors for cyber criminals to take down websites, steal sensitive data, make a profit, and more. Much of this work is done with the help of malicious software, or malware, which seeks to exploit various software vulnerabilities.

Within the realm of exploit techniques, many defenses have been developed in order to mitigate exploits. Stack canaries have been implemented to mitigate buffer overflows, and ASLR has been enforced to prevent ROP exploits. But unfortunately, because exploit mitigations are primarily developed as responses, new exploits have free reign until defenses are developed.

Thus, we aim to provide a solution that allows for the accurate detection of both existing and new exploits. One past solution involved exploit signature detection. Specific exploits can be identified by monitoring evidence such as long strings or repeated requests. But, the disadvantage of signature based exploit detection is that many systems rely on manually generated databases. While effective, when faced with new exploits, it is far too slow to identify and generate a signature for detection [12].

Efforts to automate detection and signature generation have resulted in dynamic taint analysis. This technique allows for the automated detection of many software exploits, and has potential for detecting new exploits, but the solution is not entirely effective when dealing with privilege escalation exploits that go beyond a user-level process [12].

We present a Control Flow Integrity (CFI) plug-in built for PANDA [7], a dynamic analysis platform that can achieve a system-wide CFI which allows us to dynamically identify when a vulnerability is exploited. Since most software exploits (e.g. stack overflows, ROP, return-to-libc) rely on some form of control flow manipulation, being able to monitor control flow from a whole-system standpoint will aid greatly in identifying generic exploits that reach beyond user level processes [13]. On a large set of malware recordings, the CFI plugin we seek to develop will allow us to obtain a reasonable measurement on how common privilege escalation exploits are in malware. To enforce CFI, we obtain contents of a memory access, hook DLLs and analyzing stack contents which has been discussed in detail by Dolan-Gavitt et al.[8].

2 Problem Motivation and Solution Overview

2.1 Panda

PANDA is an open-source Platform for Architecture-Neutral Dynamic Analysis [7]. It allows us to record execution from whole virtual machines running applications and then conduct an iterative approach for analysis on these replays.

PANDA is built over QEMU which is able to emulate a large set of architectures and uses QEMU intermediate language(IL) to achieve this. PANDA uses this IL to generate LLVM instruction upon which it can perform architecture-neutral analysis.

PANDA uses plugins to perform its analysis, they are easy to load and can execute callbacks at various points of QEMU emulator execution. For our plugin, we make use of the following callbacks: `PANDA_CB_BEFORE_BLOCK_TRANSLATE()` and `PANDA_CB_AFTER_BLOCK_TRANSLATE()` are invoked before and after guest code is translated into basic blocks. Similarly, `PANDA_CB_BEFORE_BLOCK_EXEC()` and `PANDA_CB_AFTER_BLOCK_EXEC()` are invoked before and after execution of a basic block.

2.2 Portable Executables

Portable Executable (PE) is the file format of Microsoft executables, object codes and DLLs. It contains necessary information, references to imported symbols, exported DLLs and resource management data so that the windows operating system is able to manage and run the executable.

PE files contain various headers that tell windows how it should to map the file into memory. The executable also contains regions that point to different sections in memory such as the text section which point to machine code, the data section which point to binary representation of data and the relocation section which identifies lines of code that need to be handled.

PE executables begin with a DOS Header which point to the PE header, that points us to other sections and data for analysis. PE headers also hold the offset addresses of loaded libraries or DLLs which are linked to the executable.

We can parse the PE header to get a list of export addresses and relocation addresses that are later used to build process and module whitelists. We also parse the text section to get a list of valid call and jump addresses of the executable if it did not contain a relocation table. The generated whitelists are checked later during execution of programs in a replay to detect violations.

3 Punctual OS View Extraction(PVE)

The PVE component builds an overview of the operating system from virtual machine and extracts process, module and thread information that is necessary for enforcing CFI and accurately determining the exploit context.

3.1 Global Data Structure Identification

Global data structure identification is a key component required to reconstruct OS views as it contains information that can be used to traverse processes, modules and threads. PANDA has helper functions that can interpret this global data structure and return guest OS semantics.

3.2 Process Identification

Process identification on PANDA is achieved using callbacks. We invoke `PANDA_BEFORE_BLOCK_EXEC()` to return a basic block which is either the first block in the process or a block traversed using a call or a jump. Once we retrieve a block, we can get details about its process using the `get_current_process()` function in PANDA. This function extracts the current thread details from the current execution context by following the path `KPCR->PRCB->_KTHREAD->_KPROCESS`. The various modules used by the process are then extracted from the guest memory by information provided in the _KPROCESS data structure.

3.3 Module Identification

Modules can be identified by keeping track of processes. PANDA provides a helper function `get_libraries()` that return a list of modules loaded by each process. When a new process is identified, we use the helper function to retrieve the list of modules which will be checked against predefined whitelists.

3.4 Thread Stack Layout Identification

Threads can be classified into three types:

1. User level thread - These threads work with the privilege of the user. Each user thread has an explicit stack assigned to it in user space.

2. Kernel level thread - These threads work with the privilege of the kernel. They are assigned a separate stack that is linked with the thread's user level stack.

3. User managed thread - These threads are created and managed by user applications. They belong to user level and share the same shadow stack with user level threads.

We maintain two global maps, one for user level threads and another for kernel level threads. A thread's stack is referenced using its process ID and thread ID after selecting the appropriate stack map using the current privilege level of execution. The algorithm used is described in 1 and draws on the Thread Stack Layout Identification algorithm developed by Prakash et al.[13].

Each of the above threads have to be identified correctly by the CFIC component in order to select the precise shadow stack to act upon. To differentiate between user level and kernel level threads, a helper function within PANDA, `panda_in_kernel()` is used. The current execution level is retrieved by this function from the operating system's state using the flags register. The

Algorithm 1 Algorithm used to pick the correct shadow stack to which addresses have to pushed and popped.

```
1:  procedure GETCURRENTSTACK(CPUState ENV)
2:      Thread ← getCurrentThead(ENV)
3:      new_thread ← FALSE
4:      current_stack ← FALSE
5:      if panda_in_kernel(ENV) == KERNEL then
6:          if Thread ∈ kernel_stacks_list then
7:              new_thread ← TRUE
8:              current_stack ← newStack()
9:              kernel_stacks_list.add(current_stack)
10:     else
11:         if Thread ∈ user_stacks_list then
12:             new_thread ← TRUE
13:             current_stack ← newStack()
14:             user_stacks_list.add(current_stack)
15:     return current_stack
```

lower two bits of the flags register contain the I/O privilege level. A value of 0 indicates that the current privilege level is that of the kernel. Otherwise, the user privilege level is used to select the shadow stack.

Once the privilege level is identified, the thread ID is retrieved by traversing the path `KPCR->PRCB->CurrentThread->Cid->UniqueThread` and the process ID is retrieved by traversing the path `KPCR->PRCB->CurrentThread->Cid->UniqueProcess` [3]. A combination of process ID and thread ID is used as the shadow stack identifier.

Our implementation does not currently support Fibers. Since, fibers run in the context of threads that schedule them, they share the same shadow stack as the user level thread that created them [2]. When a new thread is created, its execution stack can overlap the stack base of another stack. Such a scenario only occurs when the previous thread has terminated. Whenever the PVE component encounters a new thread, the stack limit can be checked for overlap with the active threads of that process. In case of overlap, the previous thread is deleted.

4 Control Flow Integrity Enforcement (CFIC)

A lot of modern exploit techniques work by altering the normal control flow of a program by manipulating control transfer addresses in scenarios such as `call` instructions, `jump` instructions, and function pointers. Using PANDA's callback architecture, regions where such control transfers can occur are monitored. The current model for enforcing control flow integrity is based on the following observations:

1. The majority of points to which control is transferred can be statically predetermined from the executable. Targets of `call` and `jump` instructions can be determined before hand.

2. A `ret` instruction should return to the address succeeding the `call` instruction that invoked the function containing the `ret` instruction.

Based on observation (1), whenever a call or a jump instruction is encountered, the target address for the control transfer is checked against the whitelist. If the target address is not found, a potential CFI violation is reported. In order to implement observation (2), two shadow stacks are maintained per executing thread in the system. Whenever a call instruction is observed, the return address is pushed to the appropriate shadow stack of the currently executing thread. When a `ret` instruction is encountered, the target address of the `ret` instruction is popped from the appropriate stack. If the target address is not found, a potential violation is reported at that `ret` instruction.

The rest of the section explain the various challenges and internal details of the CFIC implementation.

4.1 Target Whitelist

The addresses in the export table and relocation table of the executable binary constitute our module whitelist. When the loader is not able to load a binary at its default location, it performs relocation. The loader then refers this relocation table and recomputes all addresses of entries in the table. Indirectly addressable code must be relocatable. Similarly, export tables contain functions that a given module exposes for use by the other modules. Addresses of such functions are resolved at runtime based on the actual load address of the dependent modules. Therefore, entries of the relocation table and the export table of a module together form valid branch targets for a module.

We additionally parse the binary if its relocation table is paged out and extract all the valid target addresses so that they can be added to the whitelist.

We maintain a file for each process, which has all relocation and export table entries for every module required by the executable. When a module is loaded, we check the whitelist cache for the corresponding address of the module. If the address is not present, we report this as a control-flow violation.

One major limitation to our approach is the memory overhead caused by `kernel32.dll` and `ntdll.dll`. These modules have a lot of export table entries and are included in most executables. Since, we maintain a whitelist file for each process, this results in the huge overhead.

4.2 Branch and Jump Statements

A jump table is an array of function pointers of jump instructions. We can get these pointers by parsing the relocation section of a PE file. For every entry in the relocation table, we check if this is an entry point into the text section. If so, we treat it as a valid function pointer and add it to the whitelist.

If there is no relocation section, the executable is analyzed instruction by instruction to check for valid jump addresses. We only receive relative offsets during this phase. When the executable is later encountered within the replay, the target address is calculated by adding the offset to contents of the IP register. The target address is then checked to see if it is present in the whitelist. If not present, the control transfer instruction is flagged as a potential violation.

4.3 Shadow Call Stack

Two shadow stacks are maintained per thread - one for user mode execution and another for kernel mode execution. PANDA helper function named `panda_in_kernel()` is used to differentiate between kernel and user execution mode. When the `PANDA_AFTER_BLOCK_TRANSLATE()` callback identifies that the current block ends with a `call` instruction, the `PANDA_AFTER_BLOCK_EXEC()` callback is notified to push the PC value after the current block to the thread's shadow stack.

When a `ret` instruction execution is encountered by the `PANDA_AFTER_BLOCK_EXEC()`, the succeeding block's `PANDA_BEFORE_BLOCK_EXEC()` callback is alerted. The topmost entry is popped from the shadow stack. If the address is not present in the shadow stack, a potential CFI violation is reported.

5 Testing and Evaluation

5.1 Testing

We created two replays that are performing a basic buffer overflow exploit and a ROP attack and also one benign replay, one where a calculator is opened up as calc.exe.

We test for correctness with these two replays that are exploit buffer overflow. We run the plugin against the benign replays and also against replays that have the vulnerable program but not exploited to test for false positives.

5.1.1 Creating replays for testing

PANDA requires the virtual machines to be in the QCOW format. To create replays for testing, we use a Windows 7 32-bit VM. Once the VM has started up, PANDA drops into a QEMU terminal where we can pass more commands to PANDA. We send the *begin_record* flag and *end_record* flags to generate replays.

All replays generated for the testing begin recording and immediately stop recording after completing the necessary executions for the replay. Every recording captures only one malicious or benign application. These recordings are kept simple so we can achieve a baseline heuristic for testing our plugins.

5.1.2 Running the replays with CFI plugins

We make the following assumptions while testing:

1. We first run PANDA's `asidstory` plugin to retrieve the PIDs of processes we want to test for control flow integrity. The retrieved PIDs are then provided to the plugin as a runtime argument.

2. The vulnerable program is present in the file system. This assumption is made so that we build the whitelist.

3. Using python pefile library to build the export table entries for all executable an parsing this to build a part of whitelist.

4. The various DLLs within a system is available for access outside the system. This will enable us to load.

5.2 Evaluation

The shadow_stack_check plugin identifies CFI violations on both the buffer overflow exploit replay and the ROP exploit replay. Once a CFI violation is detected, the replay is allowed to proceed to check for additional CFI violations. The plugin is also run on a benign replay of calc.exe. This replay does not throw any CFI violations.

6 Limitations

Several of the data structures used in extracting the process and thread details from the guest OS are internal to the kernel. The layout of these data structures change greatly between Windows versions and even between kernel builds [5]. As a result, writing a general plugin for multiple versions of windows itself does not scale well. Thread and process lifetimes are not being checked.

Position independent code is placed somewhere in memory and it executes properly regardless of the absolute address. We are not handling such code. Also, dynamically generated code is not handled by our CFI.

7 Related work

Static signature based detection: Early attempts to detect exploits in malware started with signature identification. Virus scanners use byte code patterns characteristic to malicious code as a heuristic for identifying malware. While effective, this approach is far from comprehensive. As the solution is static in nature, signature identification fails to detect new kinds of malware and any form of obfuscation will bypass detection.

Dynamic Analysis: Willems et al. [14] present CWSandbox, a dynamic analysis tool that seeks to automate the malware analysis process. The tool was run against over six thousand binaries with significant success, but it has shortcomings. The CWSandbox relies on monitoring system calls, but it does not collect enough information for monitoring privilege escalation exploits that impact a system beyond the user level process. Anubis [4], another dynamic malware analysis program, also relies on monitoring system calls.

Control Flow Integrity verification: More recently, researchers have turned to CFI verification for exploit detection. Prakash et al. [13] observed that most modern exploit techniques like return-to-libc, use-after-free, and ROP violate control flow integrity by manipulating return addresses. This observation holds true for privilege escalation exploits as well. In monitoring system-wide CFI, generic exploits as well as emerging exploit techniques can be detected for further analysis.

A system-wide CFI checking plugin called TotalCFI [13] was implemented for DECAF [9], another dynamic whole system analysis platform. TotalCFI attempts to identify OS entities like processes and threads directly from the hypervisor. CFI monitoring through PANDA allows for the large scale detection of exploits from a malware corpus. A new thread stack identification algorithm helps identify CFI violations with minimal access to guest memory improving performance. Hardware performance counters that collect allowed control flow transfers were used in CFIMon[15] for the x86 architecture, which uses. The branch tracing store mechanism found in processors is then used to collect and analyze runtime traces that detect CFI violations.

Our approach of monitoring CFI with PANDA differs from previous efforts by following a dynamic analysis path [6, 11] and leveraging whole-system record and replay. A CFI violation indicates that an exploit has occurred, not what vulnerability has been used to exploit the system. However, the indication of a CFI violation can still be used to investigate the vulnerability.

Our implementation in PANDA allows for the repeated execution of a piece of malware with instruction-level accuracy [7]. The entire system state is captured which also aids the further analysis of malware after the initial detection. With the ability to replay execution with such detail, CFI violations can be identified with a high level of accuracy. Overhead concerning the replication of malware execution is also completely eliminated with the use of PANDA.

Program Shepherding: An alternative to detecting malware exploits through CFI violations would be to use program shepherding [10]. Shepherding has benefits such as restricting execution privileges on the basis of code origins, restricting control transfers based on instruction class, source and target while also preventing code from escaping any sandboxing policies. The restriction of execution based on code origins aids in preventing a program from mistakenly executing user data as code. Doing so severely limits the effectiveness of memory corruption attacks. Given that program shepherding can monitor all control flow transfers, sandbox enforcement cannot be bypassed, and control flow transfers are restricted, exploits can be detected by identifying the blocked violations to program control flow.

8 Future Work

Our implementation of CFI through plugins do not account for Fibers. Fibers share the same resources and works within a user level thread.

Currently, two separate plugins are used, one for extracting runtime modules and export addresses, and another one to enforce CFI though shadow call stacks.

Process termination, and thread termination is not taken care of. Implementation of process and thread termination will make the shadow call stack implementation much more accurate.

8.1 Special Control Flows

Even though majority of the control transfers have strict pairing between `call` and `ret` instructions, certain boundary conditions tend to break this strict pairing.

Handling of Exceptions: Exceptions allow handling of unforeseen or unusual conditions related to the operation of a program. The exception handler or the `catch` block is relocatable, and hence appears in the relocation table as an entry. Special care has to be taken while handling exceptions as they have unexpected jumps and addresses popped from a shadow stack need not match with the target address of a `ret` instruction.

Kernel mode to User mode call backs: In Windows, NTDLL has a few entry points through which the kernel can invoke certain functionality on behalf of the user mode. Some of the NTDLL APIs that contain such entry points are: *KiUserExceptionDispatcher, KiUserApcDispatcher, KiRaiseUserExceptionDispatcher* and *KiUser-*

CallbackDispatcher [1]. These control transfers are different from the usual where the transfer from user mode to kernel mode occur through `syscall, int` instructions and back to user mode occur using `sysexit, iret` instructions.

We do not consider the above mentioned special scenarios, and therefore additional false positive violations can be thrown by our implementation.

9 Conclusion

In this paper we present PANDA plugins that are a proof of concept of system-wide control flow integrity enforcement for malware exploit detection. PANDA's callback architecture and API are used to extract guest OS semantics that are used by the CFIC component of the plugins. The plugins were evaluated against benign replays as well as replays containing attacks such as buffer overflows and return-oriented programming.

References

[1] A catalog of ntdll kernel mode to user mode callbacks. `http://www.nynaeve.net/?p=200`.

[2] Fibers. `https://msdn.microsoft.com/en-us/library/ms682661.aspx`.

[3] Windows 7 sp1 x86 vtypes. `https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/overlays/windows/win7_sp1_x86_vtypes.py`.

[4] BAYER, U., HABIBI, I., BALZAROTTI, D., KIRDA, E., AND KRUEGEL, C. A view on current malware behaviors. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (Berkeley, CA, USA, 2009), LEET'09, USENIX Association, pp. 8–8.

[5] CHAPPELL, G. Ethread. `http://www.geoffchappell.com/studies/windows/km/ntoskrnl/structs/ethread/index.htm`.

[6] DIATCHKI, I., PIKE, L., AND ERKK, L. Practical considerations in control-flow integrity monitoring. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (March 2011), pp. 537–544.

[7] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (2015), PPREW-5, ACM, pp. 4:1–4:11.

[8] DOLAN-GAVITT, B., LEEK, T., HODOSH, J., AND LEE, W. Tappan zee (north) bridge: Mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security* (2013), CCS '13, ACM, pp. 839–850.

[9] HENDERSON, A., PRAKASH, A., YAN, L. K., HU, X., WANG, X., ZHOU, R., AND YIN, H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), ISSTA 2014, ACM, pp. 248–258.

[10] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.

[11] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (Dec 2007), pp. 421–430.

[12] NEWSOME, J. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

[13] PRAKASH, A., YIN, H., AND LIANG, Z. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (2013), ASIA CCS '13, ACM, pp. 311–322.

[14] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security Privacy 5*, 2 (March 2007), 32–39.

[15] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2012), DSN '12, IEEE Computer Society, pp. 1–12.