Code Shredding: Byte-Granular Randomization of Program Layout for Detecting Code-Reuse Attacks

Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, Takeo Hariu
NTT Secure Platform Laboratories, NTT Corporation
9-11, Midori-Cho 3-Chome, Musashino-Shi, Tokyo 180-8585, Japan
{shioji.eitaro, yuhei.kawakoya, makoto.iwamura, takeo.hariu}@lab.ntt.co.jp

ABSTRACT

Code-reuse attacks by corrupting memory address pointers have been a major threat of software for many years. There have been numerous defenses proposed for countering this threat, but majority of them impose strict restrictions on software deployment such as requiring recompilation with a custom compiler, or causing integrity problems due to program modification. One notable exception is ASLR (address space layout randomization) which is a widespread defense free of such burdens, but is also known to be penetrated by a class of attacks that takes advantage of its coarse randomization granularity. Focusing on minimizing randomization granularity while also possessing these advantages of ASLR to the greatest extent, we propose a novel defensive approach called code shredding: a defensive scheme based on the idea of embedding the checksum value of a memory address as a part of itself. This simple yet effective approach hinders designation of specific address used in code-reuse attacks, by giving attackers an illusion of program code shredded into pieces at byte granularity and dispersed randomly over memory space. We show our design and implementation of a proof-of-concept prototype system for the Windows platform and the results from several experiments conducted to confirm its feasibility and performance overheads.

1. INTRODUCTION

Attacks that alter the control flow of a running program by corrupting pointers on memory have been a major threat of binary-layer software for many years and are still prevalent today. Such type of attacks is an attractive choice for attackers because they often give them chance to run arbitrary computation on the target host upon success. Typically, these attacks are conducted by somehow modifying the code-pointing address to point to the code the attacker wishes to execute. Traditionally, the code to be executed has often been prepared by injecting it into writable memory regions such as stack or heap using vulnerabilities such as buffer over-flows. However, as code injection has become increasingly harder to perform with the spread of data execution prevention features, code-reuse attacks which jumps instead to code already existing on memory has become more popular. Examples of code-reuse attacks include the classic return-to-libc attacks as well as state-of-the-art variants of return-oriented programming attacks.

To counter this threat, numerous defenses have been proposed, and some of which are often categorized as control-flow integrity checking, sandboxing, software diversification, and so on. Apparently, however, most of them have not been very successful at being widely adopted for general use. This is most likely due to their strict restrictions which hinder their smooth integration into the development and deployment processes of target software. An example of such restriction is a requirement for recompilation with a customized compiler, which requires the proposal to be adopted by major compilers if it were become a standard. Another major problem is the integrity issues caused by changing the original instructions to be executed. This topic is discussed more, for example, in the literature[13] which describes the difficulties faced by software diversification approaches for massive-scale software distribution.

Nevertheless, there is one notable exception among these, namely ASLR (address space layout randomization). ASLR’s main idea is to randomize the locations of various important memory objects, which include the load addresses of program images. As a result, it makes attackers much more difficult to locate the desired fragment of code to which they want to change the control flow, thereby securing against attacks that specify absolute addresses. ASLR was first introduced as a third-party Linux patch by the PaX project[28], and has recently been aggressively adopted by popular OSes such as Windows[1], Linux and Mac OS, and also by performance-sensitive mobile platforms such as iOS and Android[2].

One of the most contributing factors for the success of ASLR is most likely its simple nature, imposing surprisingly weak requirements on the target software; basically, it only shifts the location of a program image on memory thus its contents are kept unchanged besides some addresses requiring relocation. Also, it is notable that the sequence of instructions to be executed is not changed at all, which preserves program semantics perfectly. However, ASLR is known to be vulnerable against some type of attacks, such as attacks combined with memory address disclosures, or partial overwrite attacks[6]. This is due to the fact that ASLR has a coarse randomization granularity but this property is so closely related to its advantageous properties that it is intrinsically difficult to fix.

Based on this observation, by focusing on minimizing randomization granularity while also possessing these advantages of ASLR to the greatest extent, we propose a novel defensive approach called code shredding, a defensive scheme based on a simple idea of embedding the checksum value of a memory address as a part of itself. Code shredding provides randomization at byte-granularity, or in other words the finest randomization granularity possible, and is also expected to minimize recompilation requirements and incur no changes to the executed instruction sequence. For avoiding con-
fusion, it is worth emphasizing first here that code shredding does not actually shred code. This approach relies on combination of code replication and destination restriction of control transfer instructions, to achieve pseudo-dispersion of code. The fundamental idea of our approach is based on self-validating addresses, which encodes addresses by calculating the checksum of an address and embed the result within the address itself. With an appropriate design and implementation, a defense using this approach will give an attacker an illusion that the code to which they wish to transfer the control flow is shredded into pieces at byte granularity and are randomly dispersed over memory space. Our proposal is not without any drawbacks of course, and we admit that it may incur considerable amount of runtime overhead on time and space. Though we are still in a status of seeking for ways to alleviate these overheads, we make a proposal of an additional sub-technique called code mirroring.

We have implemented a prototype of this proposal for the Windows platform using the Pin dynamic binary instrumentation framework. Experiments are conducted to evaluate its security effectiveness and performance. The contributions presented in this paper are summarized as follows:

- We present code shredding, a novel scheme for detecting invalid control-flow transfers, based on the following two ideas:
  - self-validating address: allows verification of memory addresses by encoding addresses in such a way that its checksum value is embedded as a part of itself.
  - code mirroring: allows an encoded address to be treated as a real addresses by actually mapping multiple copies of the image on memory.
- We present our design and implementation of a proof-of-concept prototype of code shredding system for the Windows platform. The results of our brief experimental evaluation suggests its feasibility concerning security and compatibility, and also clarifies some problems which need to be addressed in the future.

The rest of this paper is organized as follows: Section 2 describes the problem, Section 3 and 4 explains in detail the design and implementation of our proposal, Section 5 describes the evaluation procedure and the result, Section 6 discusses our limitations and related researches, and finally Section 7 concludes the paper.

2. PROBLEM DESCRIPTION

In this section we describe the type of attack we focus on, investigate the problems confronted by existing defense approaches, and state the problems which we aim to solve in this paper.

2.1 Control-Flow Hijacking Attacks

The type of attack which we focus on is what changes the control flow of a running program by modifying code pointers. We define a code pointer as any data on memory that is meant to contain an address which points to, or will eventually point to, a location on memory where executable instruction exists or will exist. Examples of code pointers include return addresses of function calls or function pointers. The goal of attackers is to execute their desired code which is a malicious sequence of instructions that triggers an attack. This can be achieved by either code injection or code reuse. Code injection writes to memory the code prepared by the attacker. However, as defensive measures such as NX-bit or DEP are becoming more common, code injection is becoming harder to perform. Code reuse, on the other hand, is not affected by such defenses because it uses already-existing codes on memory with execute permission. A famous example of such attack is return-to-lib attack[4], but recently a more generalized form of such attack called ROP (Return-Oriented Programming)[5] has become a large threat and is evolving continuously, with its variants or automation techniques being constantly presented.

2.2 Existing Defenses: ASLR

Although ASLR makes it much more difficult to write stable exploits, it is not a perfect measure even when combined with DEP or other defenses. It is in fact possible to evade ASLR when certain conditions are met. In this paper we focus on the two types of attacks that defeats ASLR, namely, attacks using address disclosure and partial overwrite attacks. ASLR-bypass using address disclosure is becoming very common, while partial overwrite attacks are relatively not so common due to its difficulty of application. However, because it shares the same intrinsic problem with address disclosure, and it in fact has been used for exploiting the famous ANI vulnerability[25], we refer to it as our motivating example. Also, as the number of systems hardened with ASLR increases, advanced exploitation techniques like partial overwrite may start to become more common in the future. Next, we describe these two attacks in more detail.

2.2.1 Attacks with Address Disclosure

Memory disclosure is a software flaw such that the content of memory is leaked to the attacker. It may occur, for example, when a format string vulnerability is exploited. Address disclosure is its special case, where a pointer is disclosed to the attacker which becomes a problem when combined with other vulnerability such as buffer overflow that can write to code pointers. Let $a$ represent the address where instruction $A$ is located on memory, and let $x$ represent the address of an arbitrary instruction $X$ in the same image. Under the assumption that the attacker has the same program image as the target program, upon knowing the address $a$, the attacker can calculate $x = a + d$ using the relative distance $d$ between $A$ and $X$ derived from the attacker’s own copy of the program image. Thereby the attacker is able to determine the address of an arbitrary instruction in this program image.

2.2.2 Partial Overwrite Attacks

On the other hand, a partial overwrite attack does not even require disclosure of concrete address values. All it requires is the knowledge that the overwriting target is an address pointing to a code in the program image. The idea behind this attack is that a typical ASLR implementation only randomizes the upper bits of address (e.g., Windows 7 32-bit randomizes bits 17 - 24) likely due to memory allocation restriction or performance reasons. Therefore, even if the upper bits are randomized, by overwriting the lower bits of an existing address, the attacker is able to specify an address within the possible range. Figure 1 describes an example of such attack. Here the stack holds a return address pointing to 0xKRR12340 where $R$ denotes the byte randomized by ASLR. Traditionally, ASLR has been effective in such a way that, if the attacker does not know the value of $R$, this byte cannot be overwritten, and therefore the whole address cannot be overwritten. However, the attacker can still overwrite the lower bytes to specify addresses of program image in the shown range without modifying $R$, and make the return address point to an address in range 0xKRR10000 - 0xKRR1FFFFFF without any knowledge of $R$. Note that this is feasible because the byte order of IA-32 is little endian, i.e., an address is stored on memory lower bytes first.
2.2.3 ASLR Problems

One of the root causes of the attacks described above is valid against ASLR is that the attackers are able to relate an address to another by referring to the same program image of their own. One trivial way to prevent them is by making the software publicly unavailable, since these attacks must assume that the attacker possesses the program image, but such measure is unrealistic in most cases. Here we define the granularity of randomization as the size of program content whose location on memory can be uniquely determined from an address disclosed to the attacker. ASLR provides randomization granularity at the size of program image, because any leaked address within the code immediately reveals the address of an arbitrary code on the image. In this sense, having finer granularity is more preferable because it will limit the attacker’s ability to relate addresses together, and ideally, having byte granularity will make such attacks impossible.

ASLR also carries several other problems. One of such is not having enough randomization entropy on systems with 32-bit memory space, making it vulnerable against brute force attacks\[14] in some cases. Also, many of the ASLR-bypass exploits in the wild to date leverage the fact that ASLR is often an opt-in defence, and reuses code on ASLR-disabled modules. Brute-forcing is likely to be mitigated by the future transition to 64-bit systems, and regarding the opt-in problem, non-ASLR enabled binaries are on the decrease as observed in the case where Firefox is no longer accepting plug-ins not supporting ASLR\[26]. It is also worth noting that Microsoft’s EMET, which is a security add-on package for Windows including ASLR-forcing feature, has recently been officially supported\[27], which implies that ASLR is now applicable to certain software not previously assured to be compatible with it. However, the problem with address-relations is an intrinsic property of ASLR, and is unlikely to be mitigated by such improvements in the future, suggesting the need of exploration for post-ASLR measures.

2.3 Existing Defenses: Other Defenses

Since the problem of control-flow hijacking has been around for a long time, there are numerous other approaches for defending against it. For example, there are approaches for randomizing at finer granularity such as at function granularity\[8], but it still suffers from address disclosure attacks. There are also other approaches that offer randomization at finer level\[9, 10, 11, 12] such as at instruction level, but they generally require recompilation from the source code, or changes executed instructions which may cause integrity problems. Some of these are discussed and compared to our proposal in Section 6.

2.4 Problem Summary

We conclude this section by describing the threat model and the properties which we aim to incorporate into our proposal.

The Threat Model:

- The execution environment is assumed to have data execution prevention system or equivalent enabled, or in other words all writable regions on memory are non-executable.
- During execution, the attacker somehow successfully overwrites a part or whole of the content of a code pointer to an arbitrary value.
- Prior to the rewriting, the attacker is able to obtain a leaked address within the program image. It can be any address except for the address overwritten to, mentioned above.

Desired Properties:

- Security: Under the threat model presented above the ability to infer the location of instruction to execute from the leaked address should be limited to the greatest extent, i.e., having the finest randomization granularity possible.
- Compatibility: The requirement for recompilation should be minimized, while preserving the integrity of the original program as much as possible.

Note that other aspects not listed above such as those concerning the effects on execution environment or performance will not be our main focus in this paper although they will be mentioned briefly.

3. DESIGN

To solve the problems stated above, we propose a novel defensive approach, code shredding, which detects invalid control-flow transfers caused by code pointer modification. In this section we describe its overview and design. Briefly described, this scheme consists of (1) conversion of addresses into specially-encoded self-validating addresses, and (2) validation of target addresses used in control flow transfer instructions at their execution. Below we describe these procedures in more detail.

3.1 Self-Validating Address

A self-validating address is simply generated in the following way:

- perform a checksum calculation on the bits at the fixed position of the address (we refer to these bits as input bits), and
- place the checksum value as the bits at another fixed position of the same address (similarly, checksum bits).

An address is considered valid if the calculated checksum from the input bits matches the checksum bits. It is called self-validating because the information needed to validate an address is self-contained, as opposed to other methods that must compare the address with other (e.g., saved) values. Unless the attacker knows how to perform the computation, it is impossible for them to generate, with enough success probability, a valid address that can be used for code-reuse attacks. In addition, if it is impossible to infer the checksum value of any other input value from a given valid converted address, it provides byte-granularity randomization. This can be realized, for example by using a secure hash function which satisfies such property and feeding the input value combined with a random value which is hidden from the attacker.
we observe 2
be the length of input bits and
having the size of 2
illustrated in Figure 2. The actual length and position are con-
ments
erated which will be discussed later.

3.2 Conversion
To be able to distinguish invalid addresses from valid addresses,
the valid addresses must be converted in advance. There is no point
of converting an address after it has been modified by an attacker, so
it is desirable to convert it as soon as the address is generated. There
are basically two types of address generation to consider: addresses
generated before program execution starts, and those dynamically
generated after.

For those generated before, the loading of program image must
be hooked, and absolute addresses that exist in the image must be
located and converted. The locating of such addresses is possible
using the relocation information on the executable. This implies
that our proposal assumes the existence of such side information,
but this is the same condition assumed by ASLR, and is relatively
easy to satisfy thanks to the widespread of ASLR.

Addresses dynamically generated during program execution are
mainly return addresses from function calls, so call instructions
must be hooked and the return address written to the stack must
be converted. There are some other occasions, mostly application
or loader specific, where absolute addresses are dynamically gen-
erated which will be discussed later.

3.3 Validation
Validation is done just before the execution of a control flow
transfer instruction. It calculates the checksum value in the same
manner as it is done in conversion and compares the calculated
value with the checksum value embedded in the address. If they
match, the address is considered valid, and otherwise it is consid-
ered invalid. Upon detection of an invalid transfer, the execution
can be halted or be continued depending on the policy. To continue
execution after validation, the address must be converted back to
the original address where the actual code exists, but this proce-
dure can be omitted as described later.

3.4 Segments
For the following discussions, and also to make the above de-
scription more comprehensible, we introduce the notion of seg-
ments. As the result of applying this design decision, letting \(N_{inp}\)
be the length of input bits and \(N_{chk}\) be the length of checksum bits,
we observe \(2^{N_{chk}}\) consecutive blocks of memory regions, each block
having the size of \(2^{N_{chk}}\) bytes. We will refer to these blocks as seg-
ments. Each segment must be at least the size of the original pro-
gram image, for each byte in segment corresponds to each byte of
original program image. Depending on the value of checksum, any
valid address will belong to one of these segments.

Increasing the length of checksum bits increases the security
strength because addresses are distributed over larger space, low-
ering the success probability of guessing attacks. The pattern of
dispersion which designates which address belongs to which seg-
ment, can be randomized by feeding a random value as a part of its
input. By choosing a new random value, say, for every process, the
valid code will appear to be located in random segments from the
perspective of attackers, making them difficult to specify them in
their exploit codes, as shown in Figure 3.

\[
\begin{align*}
\text{Figure 2: a self-validating address} \\
\text{Figure 3: two patterns of specifiable destination addresses} \\
\text{dispersed across multiple segments, randomized differently in} \\
each execution (each square represents a byte).
\end{align*}
\]

Example.
Here we show an example, with its layout illustrated in Figure
4. Assuming 32-bit memory address space, let an address be rep-
resented by \(0xKKAXXXX\) where the input bits are bits 1 – 20
denoted by \(X'\), checksum bits are bits 21 – 24 denoted by A, and
remainder bits are bits 25 – 32 represented by K which are don’t-care
bits in this example. Let \(Hash()\) be a hash function from 20 bits space
to 4 bits space, and \(R\) be a random constant chosen uniformly random from 20 bits space. We define an ad-
dress valid if it satisfies \(A = Hash(R + X')\). Then any address origi-
ally existed in the program will, after conversion, belong to one of
the 16 segments \(0xKK000000 – 0xKK0FFFFF, 0xKK100000 –
0xKK1FFFFF, \ldots, 0xKKF00000 – 0xKKFFFFFF\) depending on
the calculated value A. For example, any address satisfying \(A =
Hash(R + X')\) = 1 would belong to \(0xKK100000 – 0xKK1FFFFF\).

\[
\begin{align*}
\text{Figure 4: memory layout of the given example} \\
\end{align*}
\]

3.5 Code Mirroring
With the approach above, a converted address does not actually
point to an executable code and thus requires that, after validation,
the address be converted back to point to the segment where code
actually exists. Here we propose an extensive idea, code mirroring for eliminating the need for this by actually placing executable code in each segment, letting the execution continue at the destination pointed by the converted address. Figure 5 illustrates the effect on execution flow when code mirroring is introduced, and diagram (c) shows dotted arrows and solid arrows overlapping with each other because now converted addresses can be treated as a real address unmodified. Consequently, the control flow will proceed in an interesting manner; a switch to a random segment occurs upon every execution of monitored control transfers instructions.

![Figure 5: (a) normal execution, (b) code shredding without code mirroring, (c) code shredding with code mirroring](image)

An obvious drawback of this scheme is the additional use of memory space, which is non-negligible because it increases exponentially with the number of checksum bits, or linearly with the number of segments created for each image. To alleviate this overhead, we believe that intra-process memory mapping which allows mapping of identical objects on virtual address space to the same physical memory space (Figure 6). This may be implemented with system APIs, such as mmap on Linux or MapViewOfFile on Windows. While such APIs are usually used for file mapping or sharing memory contents among different processes, here we use it within the same virtual address space of a single process. Note that an idea similar to intra-process memory mapping has been proposed as VMA Mirroring[30], as discussed later.

![Figure 6: intra-process memory mapping](image)

3.6 Overall View

Lastly, we show the overall flow of our system in Figure 7, for the case with single loaded image. Note that the procedures p1 and p2 enclosed by dotted lines are in trade-off relations, i.e., when code mirroring is applied p1 is required but p2 is not, and vice versa.

![Figure 7: flowchart of procedures of our scheme.](image)

4. IMPLEMENTATION

In this section we describe the prototype implementation of our proposal. It is implemented for the Windows/IA-32 platform using the Intel Pin dynamic binary instrumentation framework[7]. Pin is a process-level VMM that JIT compiles code as the targeted program executes, and allows instrumentation of additional code. Note that our implementation is used for a brief evaluation in this paper and is still an ongoing work. Some functions such as intra-process memory mapping has not been implemented yet, but it should not greatly affect basic feasibility nor performance.

4.1 Conversion

Pre-Generated Addresses

We hook the end of program image loading, and copy the loaded program image to each segments which are allocated. This is when we convert the pre-generated addresses. Generally, PE files have metadata called relocation table for finding absolute addresses hardcoded in the program image for relocation purposes, so we find these addresses by parsing the entries of the associated table which can be located from header information. Our conversion targets are code pointers, so in this implementation we make all data pointers in all segments point to the data section of the first segment. The decision to determine whether the address points to data or code is done by investigating the flags of the section of the destination. If it has a write flag, it is considered data and is not considered subject for conversion. This is shown to be not sufficient in some cases and will crash some programs, and this problem is discussed later.

Note that with this setup, we have a great property such that, although each program image is loaded at different memory locations...
their contents will be exactly the same, which implies that it can be effectively shared by intra-process memory mapping.

**Dynamically Generated Addresses**

All CALL instructions generate a return address which is pushed onto the stack when they are executed, so these instructions must be hooked for address conversion. One straightforward way for doing this is by hooking the instruction followed by the CALL instruction, and directly modify the return address stored on the stack. For performance reasons, we instrumented code that emulates a call instruction that does the conversion at the same time, avoiding the need for the second memory access.

### 4.2 Validation

The destinations of all control flow transfer instructions that refer to absolute addresses are subject for validation. On IA-32, they are absolute JMP, absolute CALL, and RET instructions, so we set hooks on all of these instructions. Note that we do not need to consider relative control flow transfer instructions because such instructions all take immediate operand on this architecture which cannot be targeted by our attack model because they cannot be overwritten. After hooking, the destination address is calculated and then validated.

### 4.3 Loader-Specific Considerations

Our prototype implementation is for the Windows system, and because we made no modifications to the loader, there are some special considerations required where interactions with the loader occur. Some of the information about internals of Windows or its API specifications mentioned below can be referred to at the MSDN website[20].

- **Handles**: Windows uses data objects called handles to refer to system objects such as processes, files, and threads. For images (or modules) which are our main concern, their handle is essentially a pointer to the base address of the image, which implies that the value of handles are relocated depending on where the image is loaded. The system often uses a handle as an ID of an object, not as an address, so it would be problematic if handles are regarded as an address for our conversion, because then the system would not be able to recognize them. Therefore, when parsing through the relocation table to look for pre-generated addresses, if an address points to the base address, we leave it unconverted as it is most likely used as a handle. For the same reason, when some Windows APIs such as LoadLibrary or GetModuleHandle return handles, we leave them unconverted. Note that, on the other hand, it may have a side-effect causing a problem if handles are used as addresses.

- **DLL Entrypoints**: DLLs often (though are not required to) have an entrypoint function defined, which are called on creation/termination of threads and processes, and also on calls of LoadLibrary/FreeLibrary APIs to perform initializations and cleanups. The absolute addresses of these entrypoint functions are stored in the internal database of the loader, and they point to the image loaded at the original base address, causing false-positives when they are dispatched. Therefore, when image loading is hooked, we parse through the loader database which is accessible from PEB(Process Environment Block), and convert these entrypoint addresses as soon as they are registered.

- **GetProcAddress**: This API takes as its argument a module handle and a procedure name, returning the address of the specified procedure. However, the returned address turns out to be an unconverted address, and this is most likely because of how functions are exported in Windows. The address is probably calculated by adding the base address of the image and the RVA(Relative Virtual Address), the offset from the base address, obtained from the export table. To avoid this problem we hook the returning from of this API and convert the return value.

- **GetModuleHandleEx**: This API is used to obtain a module handle from the argument pointing to the name of image name to look for. However, when it is called with the dwFlags argument with an appropriate bit set, it accepts an arbitrary address instead of a string pointer, and returns a handle of the module that contains this address. If the address passed to this API is a converted address, the loader would return a value indicating that the lookup has failed, causing a semantic change to the program, if not a crash. Therefore, we hook the call to this API, check the dwFlags, and if its 0x4 bit is set, we convert back the address specified as the argument lpModuleName.

- **Restriction of Pin**: Although this is not an issue directly related to the loader, we mention here that Pin has a limitation where it becomes active only after kernel32.dll, kernelbase.dll, and ntdll.dll have been loaded. Thus, it misses some of the initial instructions executed by these DLLs, as mentioned in the paper published by its developers[22]. This is most likely the cause for false negative and crashes when trying to apply our system to these DLLs. Therefore, in our prototype we will always be excluding these DLLs from our target.

### 4.4 Hash Calculation

For this prototype we used a simple hashing algorithm which just adds up each byte of the input value, adds a randomly generated byte value, and takes the modulo $2^{N_{ch}}$ of the result, where $N_{ch}$ is the number of checksum bits. Note that such hash algorithm is not considered secure in practice, as two disclosed addresses would instantly reveal the secret random key, but for our threat model it should be enough. Also, for performance reasons we cached the pre-calculated hash values on a cache table which consumes memory size approximately equal to the memory size consumed by a set of segments created for one program image. Still, note that only one copy of this table is required, because it can be shared among all different images in the same process. Such table is feasible for 32-bit implementation, but considering that they fully consume the equal size of physical pages unlike images, and that the table size grows with 64-bit implementation, calculating hashes on-the-fly may be a better idea.

### 5. EVALUATION

In this section, we briefly evaluate our prototype from the two aspects we decided to focus on, which are security and compatibility. Other issues are discussed in the next section.

#### 5.1 Set-up for Experimental Evaluation

We have conducted simple experiments with our prototype. The environment is Windows 7 Ultimate SP1 32bit, 2-core Intel Xeon X5670 at 2.93GHz with 4.00GB RAM on VMWare Workstation 7. It should be noted that our current implementation only partially supports application to DLL images, due to some unanalyzed crashes and restrictions of Pin. The number of input bits is 24, and
that of checksum bits is 2, implying 4 copied segments are created for each image loaded for the process.

5.2 Security Evaluation

5.2.1 Address Disclosures and Partial Overwrite

Even if the attacker has gained an address pointing to a code in the program image, without knowing the secret randomization key, attacker cannot identify which segment the address of target code exists. Guessing would only allow success at the rate of once every $2^{N_{chk}}$ attempts on average, where $N_{chk}$ is the length of checksum bits. Partial overwrite would also not work for the same reason.

One may argue that, this scheme is not really byte-granular because a leaked address will reveal the checksum for all other addresses with the same input bits. In other words, using a leaked address 0xKKYAAAAA, where Y and AAAAA are checksum bits and input bits respectively, the attacker can designate all addresses 0XXYYYYAAA for arbitrary XX. However, in such a case, the related addresses are inter-modal, so that the leaked address and the addresses targeted for code reuse can safely be isolated by ASLR. The only concern is when the disclosed address turns out to match the destination address of invalid transfer. However, as ROP usually requires multiples addresses, one of each gadget, so such case should not be significant because the effect is confined to at most one.

5.2.2 Experimental Evaluation

We have also run an experiment to see if it detects partial overwrite attacks, although it is somewhat already obvious that it does. Assuming the case where there is an exploitable vulnerability in the function LoadLibraryEx of kernel32.dll which is called by Firefox 5.0.1 executable, we simulated a partial overwrite attack that modifies the lower 16 bits of the return address from this function, returning into code on the Firefox main executable. We confirmed that our prototype successfully detected this attack as expected.

5.3 Compatibility Evaluation

Here we present and analyze two examples of problematic cases we encountered during experiments, generalize and clarify the problems our proposal has, and suggest workarounds.

5.3.1 Experimental Evaluation

To test compatibility, we ran various programs on our prototype. Programs such as ftp.exe, calc.exe, mspaint.exe, and notepad.exe from the folder C:\Windows\System32, and also Adobe Reader 10.1.1(Non-protected mode), Firefox 5.0.1, Internet Explorer 8.0, Hover 6.1, etc.. were confirmed to run without problems, except for some that needed a workaround mentioned in case analysis 2. Again, it should be noted that DLLs are not supported so perfect compatibility is not yet assured. Some of the problems confronted are analyzed below:

5.3.2 Case Analysis 1: Direct computation on addresses

The first example we have is a false positive case which was caused by the following instruction sequence.

```
mov ecx, [ptr]
add ecx, 5
call ecx
```

It loads a value from the memory to the ECX register, adds an immediate value to it, and then calls a function it points to. This will likely cause a false positive because the input bits of the address are modified, but the checksum value no longer corresponds to it. Our proposal compares the checksum calculated from lower bits with that of higher bits, so when an already-generated address is directly modified, it would cause a false positive. A workaround for this problem would be to detect such occurrences by static/dynamic analysis and make it into consideration when calculating a checksum. It still cannot cover all cases, so for a perfect solution, it is needed to restrict the compiler from generating such code.

5.3.3 Case Analysis 2: Mixture of data and code

Next, we present a case which caused a crash. It occurred in a function that takes two data pointers, one holds the start address of a function pointer table, and the other one holds the end address of the same table. In the function, each function on the table is executed by incrementing the starting pointer until the it reaches the end address. The problem arises because both of these addresses are subject to conversion because the table is located in the code section despite that they are actually data. Therefore, with high probability the two addresses would point to different segments, and the function will keep reading the table beyond its end as illustrated in Figure 8.

This is another example that illustrates a problem being caused by the difficulty of separating code pointers from data pointers. To overcome this problem, the compiler must add additional information on the relocation information indicating if the address points to data or code. For this particular case, the first and the last entries of the table turned to be always zero, so we evaded this problem by skipping the conversion of addresses which points to value zero, which is unlikely to be a valid code.

![Figure 8: Pstart, Pend are pointers to the head and tail of the function table respectively. (a) unmodified execution where each function pointer on the table is executed consecutively, and (b) execution with code shredding where a reference to function pointer occurs outside of the table.](image)

5.3.4 Other Crashes

Although most programs we tested ran properly when code shredding is applied only to the main executable, there are still some DLLs that causes crashes when code shredding is applied. These cases are currently under analysis for their cause.

5.4 Runtime Performance

As our benchmarking application, we used Bzip2 1.0.5 for Windows[19], which is part of GnuWin toolsets, compiled from source
code with Visual Studio 2010. This application loads 5 images, bzip2.exe, msvcr100.dll, nturl.dll, kernelbase.dll, kernel32.dll, and the last three DLLs are not targeted due to the restriction of Pin mentioned earlier in this paper. The executed command is: bzip2 -zkf rand_X.dat, where X represents the size of used data file in bytes and is one of 100K, 1M, 10M, 100M, and 1000M. The content of data file is randomly generated by the command: head -c X /dev/urandom > rand_X.dat.

For images not being targeted with code shredding, we hooked all call instructions to perform dummy conversion procedure, and also hooked all indirect control-flow transfer instructions jumping to non-targeted images to perform dummy detection procedure to artificially introduce the overhead.

Table 1: Execution time and overheads

<table>
<thead>
<tr>
<th></th>
<th>nat(s)</th>
<th>pin(s)</th>
<th>cs(s)</th>
<th>cs/nat</th>
<th>pin/nat</th>
<th>cs/pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K</td>
<td>0.077</td>
<td>1.159</td>
<td>1.998</td>
<td>26.10</td>
<td>15.14</td>
<td>1.72</td>
</tr>
<tr>
<td>1M</td>
<td>0.311</td>
<td>1.782</td>
<td>2.882</td>
<td>9.27</td>
<td>5.73</td>
<td>1.62</td>
</tr>
<tr>
<td>10M</td>
<td>2.590</td>
<td>7.763</td>
<td>11.46</td>
<td>4.43</td>
<td>3.00</td>
<td>1.48</td>
</tr>
<tr>
<td>100M</td>
<td>25.17</td>
<td>66.92</td>
<td>96.18</td>
<td>3.82</td>
<td>2.66</td>
<td>1.44</td>
</tr>
<tr>
<td>1000M</td>
<td>254.9</td>
<td>668.0</td>
<td>931.8</td>
<td>3.65</td>
<td>2.62</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Table 2: Number of executed instructions and hook ratio

<table>
<thead>
<tr>
<th></th>
<th>num. of inst.</th>
<th>convert(%)</th>
<th>verify(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K</td>
<td>1.25 x 10^8</td>
<td>0.165</td>
<td>0.168</td>
</tr>
<tr>
<td>1M</td>
<td>1.27 x 10^9</td>
<td>0.221</td>
<td>0.221</td>
</tr>
<tr>
<td>10M</td>
<td>1.27 x 10^10</td>
<td>0.227</td>
<td>0.228</td>
</tr>
<tr>
<td>100M</td>
<td>1.28 x 10^11</td>
<td>0.228</td>
<td>0.228</td>
</tr>
<tr>
<td>1000M</td>
<td>1.27 x 10^12</td>
<td>0.228</td>
<td>0.229</td>
</tr>
</tbody>
</table>

Table 1 shows the execution time and overhead of Bzip2 for compressing each data file. Columns 2-4 show the execution time in seconds, where nat: native execution, pin: pin without any instrumentation, and cs: code shredding. Columns 5-7 show their ratio, so for example, cs/nat indicates the overhead of code shredding over native execution, expressed as ratio. The results show that as the execution time grows, the cs/nat overhead converges to around 3.6, and cs/pin overhead converges to around 1.4. The large overhead with short execution is mainly because of how Pin works, including Pin’s setup time and the initial execution JIT overhead, as shown by the pin/native Pin without instrumentation.

We also profiled the execution frequency of hooked instructions that cause overhead, i.e., the call conversion hooks (call instructions) and verification hooks (indirect control-flow transfers). Table 2 shows the percentage of such instructions executed out of all executed instructions. It shows that they both converge to around 0.228 percent for this application.

We have also ran some GUI applications mentioned in the compatibility evaluation part, and although booting took longer than usual, we were able to use them, (e.g. browse the web, edit texts) without much stress once booted. There are some Pin-dependent as well as Pin-independent optimizations which we have not yet applied, so these results are only tentative.

6. DISCUSSIONS

In this section we discuss the limitations of our work as well as related work.

6.1 Known Limitations

6.1.1 Non-Code Pointers

To change the control flow, the attacker may instead choose to modify data pointers such as flag data or user information which are referred to at conditional branching statements, but such attacks are out of scope of this paper. However, there is a case where data pointers can be indirect code pointers, which hold addresses of code pointers. This will become a problem because since data is all in the same segment, if we have an address disclosure of any of these data, all such indirect code pointers will be targeted for code-reuse attacks.

6.1.2 Memory Usage

Although consumption of physical memory space may be alleviated to a certain extent, consumption of virtual memory space may start to become an issue when larger number of copies is required to satisfy strict security demands. We expect that this problem will eventually be solved by the recent spread of 64-bit systems.

6.1.3 Dynamic Code Generation

Another type of dynamically generated addresses pointing to code, is seen in applications that generate and execute code on-the-fly as in script language interpreters. Since it is not possible to detect such generation of address without understanding the program semantics, our current approach is not applicable to such programs. This implies JIT spray attacks [21], which is a code reuse attack on dynamically generated code, cannot be prevented by our defense, although it can be claimed that JIT spray attacks are limited to special applications and such flaws should be fixed independently. In any case, for applying our proposal to programs with dynamically generated codes in general, it needs to be implemented at the application layer.

6.1.4 Dynamically Loaded Libraries

As stated in the previous section, our current implementation can target only one program image, which is the executable file image. Because most programs load tens of additional DLLs after the loading of main executable file, and because in many cases majority of executed instruction exist in DLLs, the effect of this may not be negligible. Also, the loading of DLLs is sometimes delayed until it is actually used, and because our proposal incurs considerable initial overhead, this may become a problem if it occurs too frequently during middle of execution.

6.1.5 Execution Environment

Because the Pin module itself cannot be the target of code shredding and it will be loaded to the same address space as the targeted software, the attacker may perform an attack by returning into Pin code. Therefore, it is more preferable to implement it in a more isolated layer such as the OS layer, the hypervisor layer, or even the hardware layer.

6.2 Related Works

Here we describe and compare some other research works, which share common ideas with our proposal.

Software Diversification

Software diversification changes the software and/or the execution environment in such a way that the attacker cannot succeed the attack unless he knows how it is changed. A well-known approach is instruction set randomization [3] which encodes instructions with a secret randomized key and decodes them right before execution.
making code injected by the attacker not decodable. There are other approaches such as stack reversing[32], but while such approaches are valid for preventing code injection they do not prevent control flow hijacking. Our approach is similar to software diversification in the sense that similar idea is applied the addresses themselves, making overwriting, or injection of addresses impossible.

**ROP prevention**

The recent increase in the number and complexity of ROP(return-oriented programming) attacks have lead to many proposals specialized for defending against such attacks, for example by monitoring the frequency of return instructions[31]. Recently, a method to eliminate ROP gadgets by static conversion of binaries has been proposed[23], which seems to be a quite effective and practical measure against ROP. Our method is not limited to ROP attacks and can prevent a more broad range of attacks which includes return-to-libc, ROP without returns[29], and other type of control-flow hijacking attacks that may be invented in the future.

**Pointer Encryption**

The most similar, and perhaps the most competing approach to ours would be the defensive measures based on encryption of addresses[18]. While their security depends on a solid cryptographic base instead of probability, encryption tend to cause high overhead. Also, while our approach encodes an address in a form still directly usable by instructions, such encrypted addresses require decryption before usage.

**Shadow Stack**

Another well-known approach is to keep track of the return address on a shadow stack which emulates the real stack, and upon function returns compare the return address to the saved values as in, for example ROPDefender[24]. An obvious limitation of such approach, aside from requiring extra memory for shadow stack, is that it can only defend against attacks that modify return addresses. One might argue that other types of addresses can also be checked by extending shadow stack to keep the copies of them; however, such addresses do not always stay at the same location like return addresses, and they are moved and copied to other places such as heap, stack, and of course, registers. It is hard to track these without applying a technique such as taint analysis which is extremely performance-intensive. In contrast, in our scheme the information needed to validate addresses are **self-contained**.

**Control Flow Integrity**

Control flow integrity(CFI) is an approach that generates a model of valid control flow graph through static analysis of the source code or binary at run-time, and enforces it during execution. While it allows a more strict and deterministic defense, it incurs heavy run-time overhead. Also, they tend to suffer from false positives when analysis is not sufficient due to lack of debug information which is common on COTS software.

**Other Defensive Approaches**

Code Pointer Masking[15] is an approach which is similar to ours in the sense that it is a probabilistic approach that verifies destination addresses at runtime. Just before its use, a code pointer is applied a special mask generated beforehand by OR-ing valid destinations. Applying this mask scrambles the pointer only if it is not pointing to a valid destination. The disadvantage of this approach, besides requiring source code analysis, is that as the number of valid destinations increases, the mask becomes more tolerant thus allowing more possible false negatives. On the other hand, in our approach the security strength is independent of the program content, and is controllable to a certain extent, i.e., the security strength can be increased to a desired level by increasing the number of segments as long as memory allows.

In-OS address space layout randomization is studied in the recent work[33] and it is claimed that information leakage is a more serious issue in the kernel context which motivates our work. Also, their idea of live re-randomization may be applied to complement our system for further strengthening. Also, binary stirring[34] is another recent proposal of static binary randomization, where challenges of transformation is studied, which may be inspected to further identify and fix our compatibility issues. While both of these works have advantageous features, they have randomization granularity of basic block.

**Memory Sharing**

The idea of mapping a physical page to multiple pages in virtual memory had also been proposed a while ago by the PaX project, as VMA mirroring[30] for Linux systems. The implementation details of VMA mirroring itself, however may be applicable when extending our work to Linux. Also, a similar idea in a more general and transparent manner is implemented on some virtual machine monitors. For example, KVM provides KSM( Kernel Samepage Merging) which merges common pages across multiple VMs (and also common pages within a VM). However, such method requires heavy scanning of memory and thus usually has long intervals between scans. It would be interesting to investigate if such scheme works well with our system.

**Randomized Control Flow**

Lastly we mention the work Branch-on-Random[16] which, although not a security related work, gave us an inspiration of our idea. It proposes a unique conditional branch instruction, where the condition is probability, i.e., decision to take the branch or not is determined probabilistically at the moment of branch instruction execution. It is similar to our approach in the sense that the control flow proceeds in random manner, though in our work the destination is randomly determined before execution.

**7. CONCLUSION**

In this paper we presented our novel idea for defending against control-flow hijacking attacks. This problem has been challenged by many research works in the past, but what distinguishes our work from the majority of them is the focus on making the approach satisfy not only security property, but also compatibility properties such as not requiring recompilation, code analysis, or preserving integrity of execution. This focus is derived from the insight that ASLR, which is the most widespread defense, satisfies the latter property sacrificing the former property.

The main purpose of this paper is the proposal of the overview and design of this approach, though we also implemented a prototype implementation on Pin to perform a brief evaluation on of its feasibility. We conclude that this approach is expected to be feasible, while for perfect compatibility it may require modifying the compiler, and such changes should be much simpler than other proposed methods. This work is expecting many further refinements and evaluation to be done in the future, which includes optimizations for improving runtime performance, compatibility improvements through crash analysis towards full support for dynamically loaded images. It would also be an interesting future work to investigate the advantages and requirements for implementing it on hardware, or application to Linux.
8. REFERENCES


