Using dual-mappings to evade automated unpackers

skape
mmiller@hick.org

Abstract

Automated unpackers such as Renovo, Saffron, and Pandora's Bochs attempt to dynamically unpack executables by detecting the execution of code from regions of virtual memory that have been written to. While this is an elegant method of detecting dynamic code execution, it is possible to evade these unpackers by *dual-mapping* physical pages to two distinct virtual address regions where one region is used as an *editable* mapping and the second region is used as an *executable* mapping. In this way, the editable mapping is written to during the unpacking process and the executable mapping is used to execute the unpacked code dynamically. This effectively evades automated unpackers which rely on detecting the execution of code from virtual addresses that have been written to.

Update: After publishing this article it was pointed out that the design of the Justin dynamic unpacking system should invalidate evasion techniques that assume that the unpacking system will only trap on the first execution attempt of a page that has been written to. Justin counters this evasion technique implicitly by enforcing $W \oplus X$ such that when a page is executed from for the first time, it is marked as executable but non-writable. Subsequent write attempts will cause the page be marked as non-executable and dirty. This logic is enforced across all virtual addresses that are mapped to the same physical pages. This has the potential to be an effective countermeasure, although there are a number of implementation complexities that may make it difficult to realize in a robust fashion, such as those related to the duplication of handles and the potential for race conditions when transitioning page protections.

1 Background

There are a number of automated unpackers that rely on detecting the execution of dynamic code from virtual addresses that has been written to. This section provides some background on the approaches taken by these unpackers.

1.1 Malware Normalization

Christodorescu et al. described a method of normalizing programs which focuses on eliminating obfuscation[2]. One of the components of this normalization process consists of an iterative algorithm that is meant to produce a program that is not self-generating. In essence, this algorithm relies on detecting dynamic code execution to identify self-generated code. To support this algorithm, QEMU was used to monitor the execution flow of the input program as well as all memory writes that occur. If execution is transferred to an address that has been written to, it is known that dynamic code is being executed.

1.2 Renovo

Renovo is similar to the malware normalization technique in that it uses an emulated environment to monitor program execution and memory writes to detect when dynamic code is executed[5]. Renovo makes use of TEMU as the execution environment for a given program. When Renovo detects the execution of code from memory that was written to in the context of a given process, it extracts the dynamic code and attempts to find the original entry point of the unpacked executable.

1.3 Saffron

Saffron uses two approaches to dynamically unpack executables[7]. The first approach involves using Pin's dynamic instrumentation facilities to monitor program execution and memory writes in a direction similar to the emulated approaches described previously[4]. The second approach makes use of hardware paging features to detect when execution is transferred to a memory region. Saffron detects the first time code is executed from a page, regardless

of whether or not it is writable, and logs information about the execution to support extracting the unpacked executable. This can be seen as a more generic version of the technique used by OllyBonE which focused on using paging features to monitor a specific subset of the address space[10]. OmniUnpack also uses an approach that is similar to Saffron[6].

1.4 Pandora's Bochs

Pandora's Bochs uses techniques similar to those used by Christodorescu and Renovo[1]. Specifically, Pandora's Bochs uses Bochs as an emulation environment in which to monitor program execution and memory writes to detect when dynamic code is executed.

1.5 Justin

Justin is a recently developed dynamic unpacking system that was presented at RAID 2008 after the completion of the initial draft of this paper[3]. Justin differs from previous work in that is uses hardware non-executable paging support to enforce $W \oplus X$ on virtual address regions. When an execution attempt occurs, an exception is generated and Justin determines whether or not the page being executed from was written to previously. The authors of Justin correctly identified the evasion technique described in the following section and have attempted to design their system to counter it. Their approach involves verifying that the protection attributes are the same across all virtual addresses that map to the same physical pages. This should be an effective countermeasure, although there is certainly room for attacking implementation weaknesses, should any exist.

2 Dual-mapping

The automated unpackers described previously rely on their ability to detect the execution of dynamic code from virtual addresses that have been written to. This implicitly assumes that the virtual address used to execute code will be equal to an address that was written to previously. While this assumption is safe in most circumstances, it is possible to use features provided by the Windows memory manager to evade this form of detection.

The basic idea behind this evasion technique involves *dual-mapping* a set of physical pages to two virtual address regions. The first region is considered an *editable* mapping and the second region is considered an *executable* mapping. The contents of the unpacked executable are written to the editable mapping and later executed using the executable mapping. Since both mappings are associated with the same physical pages, the act of writing to the editable mapping indirectly alters the contents of the executable mapping. This evades detection by making it appear that the code that is executed from the executable mapping was never actually written to. This technique is preferable to writing the unpacked executable to disk and then mapping it into memory as doing so would enable trivial unpacking and detection.

Implementing this evasion technique on Windows can be accomplished using fully supported user-mode APIs. First, a pagefile-backed section (anonymous memory mapping) must be created using the `CreateFileMapping` API. The handle returned from this function must then be passed to `MapViewOfFile` to create both the editable and executable mappings. Finally, the dynamic code must be unpacked into the editable mapping through whatever means and then executed using the executable mapping. This is illustrated in the code below:

```
ImageMapping = CreateFileMapping(
    INVALID_HANDLE_VALUE, NULL,
    PAGE_EXECUTE_READWRITE | SEC_COMMIT,
    0, CodeLength, NULL);

EditableBaseAddress = MapViewOfFile(ImageMapping,
    FILE_MAP_READ | FILE_MAP_WRITE,
    0, 0, 0);
ExecutableBaseAddress = MapViewOfFile(ImageMapping,
    FILE_MAP_EXECUTE | FILE_MAP_READ | FILE_MAP_WRITE,
    0, 0, 0);

CopyMemory(EditableBaseAddress,
    CodeBuffer,CodeLength);

((VOID (*)())ExecutableBaseAddress)();
```

The example code provides an illustration of using this technique to execute dynamic code. This technique should also be fairly easy to adapt to the unpacking code used by existing packers. One consideration that must be made when using this technique is that relocations must be applied to the unpacked executable relative to the base address of the executable mapping. With that said, the relocation fixups themselves must be applied to the editable mapping in order to avoid tainting the executable mapping.

An additional evasion technique may also be necessary for dynamic unpackers that monitor code execution from any virtual address, regardless of whether or not it was previously written to. This is the case with Saffron's paging-based[7] automated unpacker.

For performance reasons, Saffron only logs information the first time code is executed from a page. If the contents of the code changes after this point, Saffron will not be aware of them. This makes it possible to evade this form of unpacking by executing innocuous code from each page of the executable mapping. Once this has finished, the actual unpacked executable can be extracted into the editable mapping and then executed normally. This evasion technique should also be effective against Justin due to the fact that Justin does not trap on subsequent execution attempts from a given virtual address[3].

While these evasion techniques are expected to be effective, they have not been experimentally verified. There are a number of reasons for this. No public version of Pandora's Bochs is currently available. However, its author has indicated that this technique should be effective. Renovo provides a web interface that can be used to analyze and unpack executables. No data was received after uploading an executable that simulated this evasion technique. The authors of Saffron have indicated that they expected this technique to be effective.

3 Weaknesses

Perhaps the most significant weakness of the dual-mapping technique is that it is not capable of evading all automated unpackers. For example, dynamic unpacking techniques that strictly focus on control flow transfers, such as PolyUnpack[9] and ParaDyn[8], should still be effective. However, this weakness could be overcome by incorporating additional evasion techniques, such as those mentioned in cited work[9].

Automated unpackers could also attempt to invalidate the dual-mapping technique by monitoring writes and code execution in terms of physical addresses rather than virtual addresses. This would be effective due to the the fact that both the editable and executable virtual mappings would refer to the same physical addresses. However, this approach would likely require a better understanding of operating system semantics since memory may be paged in and out at any time.

4 Conclusion

The dual-mapping technique can be used by packers to evade automated unpacking tools that rely on detecting dynamic code execution from virtual addresses that have been written to. While this evasion technique is expected to be effective in its current form, it should be possible for automated unpackers to adapt to handle this scenario such as by monitoring writes to physical pages or by better understanding operating system semantics that deal with virtual memory mappings.

References

[1] L. Boehne. Pandora's bochs: Automatic unpacking of malware. Jan 2008.

[2] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report 1539, University of Wisconsin and Madison, Wisconsin, USA, November 2005.

[3] Fanglu Guo, Peter Ferrie, and Tzi cker Chiueh. A study of the packer problem and its solutions. In *RAID*, pages 98–115, 2008.

[4] Intel. Pin. http://rogue.colorado.edu/pin/.

[5] M. Gyung Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. http://www.andrew.cmu.edu/user/ppoosank/papers/renovo.pdf, Oct 2007.

[6] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. http://www.acsac.org/2007/papers/151.pdf, December 2007.

[7] Danny Quist and Valsmith. Covert debugging: Circumventing software armoring techniques. *BlackHat USA*, Aug 2007.

[8] K. Roundy. Analysis and instrumentation of packed binary code. http://www.cs.wisc.edu/condor/PCW2008/paradyn_presentations/roundy-packedCode.ppt, Apr 2008.

[9] P. Royal, M. Haplin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. *22nd Annual Computer Security Applications Conference*, Dec 2005.

[10] J. Stewart. Ollybone. 2006.