

**USAFA-CN-2013-457**

**Hardware Assisted  
ROP Detection Mode  
(HARD Mode)**



**NATHANIEL HART  
MICHAEL WINSTEAD  
MARTIN CARLISLE  
MICHAEL LEMAY  
RODNEY LYKINS**

**HQ USAFA/DFCS  
2354 FAIRCHILD DRIVE,  
SUITE 6G-101  
USAFA, CO 80840**

**AUGUST 2013**

***Academy Center for Cyberspace Research (ACCR)  
Department of Computer Science***

**DEAN OF THE FACULTY  
UNITED STATES AIR FORCE ACADEMY  
COLORADO 80840**

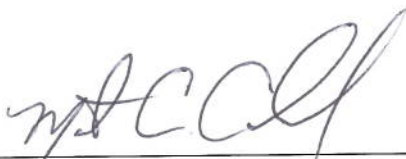


USAFA CN-2013-457

This report, "Hardware Assisted ROP Detection Mode (HARD Mode)" is presented as a competent treatment of the subject, worthy of publication. The United States Air Force Academy vouches for the quality of the research, without necessarily endorsing the opinions and conclusions of the authors. Therefore, the views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the US Government.

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the US Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is releasable to the Defense Technical Information Center (DTIC) subject to the distribution limitations indicated on the cover of this document.



DR. MARTIN C. CARLISLE  
Director  
Academy Center for Cyberspace Research

23 Sept 2013

Date



ROBERT J. KRAUS, Colonel, USAF  
Chief Scientist, Dean of the Faculty  
United States Air Force Academy

26 SEPT 2013

Date

**REPORT DOCUMENTATION PAGE**

Form Approved  
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to the Department of Defense, Executive Service Directorate (0704-0188). Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.**

<b>1. REPORT DATE (DD-MM-YYYY)</b> 16-08-2013		<b>2. REPORT TYPE</b> Technical		<b>3. DATES COVERED (From - To)</b> 20120810-20121215	
<b>4. TITLE AND SUBTITLE</b> Hardware Assisted ROP Detection Mode (HARD Mode)				<b>5a. CONTRACT NUMBER</b>	
				<b>5b. GRANT NUMBER</b>	
				<b>5c. PROGRAM ELEMENT NUMBER</b>	
<b>6. AUTHOR(S)</b> NATHANIEL HART MICHAEL WINSTEAD MARTIN CARLISLE RODNEY LYKINS MICHAEL LEMAY				<b>5d. PROJECT NUMBER</b>	
				<b>5e. TASK NUMBER</b>	
				<b>5f. WORK UNIT NUMBER</b>	
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Academy Center for Cyberspace Research HQ USAFA/DFCS 2354 FAIRCHILD DRIVE, SUITE 6G-101				<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  XXXX-XX	
<b>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> NONE				<b>10. SPONSOR/MONITOR'S ACRONYM(S)</b>	
				<b>11. SPONSOR/MONITOR'S REPORT NUMBER(S)</b>	
<b>12. DISTRIBUTION/AVAILABILITY STATEMENT</b> Publicly Releasable					
<b>13. SUPPLEMENTARY NOTES</b>					
<b>14. ABSTRACT</b> Return oriented programming (ROP) is a form of code-reuse attack employed in many modern exploitation attacks. Current defenses such as address-space randomization, structured exception handling, and memory space permissions have thus far proven only speed bumps for attackers. Utilizing new hardware capabilities in the upcoming Intel Haswell platform, we have leveraged a hardware-based approach to protect against a ROP attack. With our process, an application's and associated libraries' code segments in memory are marked non-executable and the page faults created when switching execution between pages are utilized as events during which invoke the decision engine. The decision engine is designed to examine the program's actions which caused it to attempt to pass a page boundary and report to an enforcement component which ensures the program's execution terminators. Our proof of concept decision engine examines returns that cross page boundaries and ensures that the target of a return is preceded by a call operation. Should a page transition be approved by a decision engine, the requested memory is marked executable. Otherwise, the enforcement engine will set the program counter register to zero, causing the application to crash.					
<b>15. SUBJECT TERMS</b> Return Oriented Programming (ROP), Malware, Hardware Code Exploitation					
<b>16. SECURITY CLASSIFICATION OF:</b>			<b>17. LIMITATION OF ABSTRACT</b>  None	<b>18. NUMBER OF PAGES</b>  10	<b>19a. NAME OF RESPONSIBLE PERSON</b> Maj Matthew P Ross
<b>a. REPORT</b>  U	<b>b. ABSTRACT</b>  U	<b>c. THIS PAGE</b>  UU			<b>19b. TELEPHONE NUMBER (Include area code)</b> 719-333-4118

## **Contents**

Contents.....	iv
List of Figures.....	v
Abstract.....	1
Introduction.....	1
Related Work.....	2
Contribution to Research.....	2
Implementation.....	3
Test Plan.....	3
Results.....	4
Conclusions and Future Work.....	4
References.....	5

## **List of Figures**

Figure 1. HARD Mode Schematic.....	2
------------------------------------	---

## **Abstract**

Return oriented programming (ROP) is a form of code-reuse attack employed in many modern exploitation attacks. Current defenses such as address-space randomization, structured exception handling, and memory space permissions have thus far proven only speed bumps for attackers. Utilizing new hardware capabilities in the upcoming Intel Haswell platform, we have leveraged a hardware-based approach to protect against a ROP attack. With our process, an application's and associated libraries' code segments in memory are marked non-executable and the page faults created when switching execution between pages are utilized as events during which invoke the decision engine. The decision engine is designed to examine the program's actions which caused it to attempt to pass a page boundary and report to an enforcement component which ensures the program's execution terminators. Our proof of concept decision engine examines returns that cross page boundaries and ensures that the target of a return is preceded by a call operation. Should a page transition be approved by a decision engine, the requested memory is marked executable. Otherwise, the enforcement engine will set the program counter register to zero, causing the application to crash.

## **Introduction**

Return oriented programming (ROP) stems from an advanced form of original stack-smashing techniques as described in the original 1996 Phrack article by Aleph One[1]. ROP redirects program flow into arbitrary places of program code by exploiting the return pointer placed upon the stack of a program. By having control over arbitrary return locations, ROP then has a program move from point to point within its memory space - often disregarding function entry-points or conventions of how a stack must be "cleaned up" after a function call. The section of code to where a ROP attack jumps is called a "gadget" and a series of gadgets linked sequentially to achieve a result is called a "ROP chain." This attack is also not limited to the code compiled directly into the target application - with the concept of separate compilation and pre-distributed libraries being prevalent, the code within these libraries is also a target.

Protections against ROP take either memory- or instruction-based approaches at runtime. Memory attempts have made shadow-copies of memory in order to detect memory trashing[2] or created layout randomization which creates one-time memory addresses per runtime[3]. Researchers at Columbia and Drexel even went so far as to create an instruction-set pre-processor[4] which encodes all valid processor opcodes at runtime and decodes them when the processor executes the instructions. These techniques all attempted to signal in some way an attack was occurring by simply creating a situation where the target class of exploits would simply fail.

In the general sense, a detection engine would combine decision-making capabilities from multiple locations and in near-real time create a decision on whether a requested action for a process is permissible. These actions may be as simple as accessing a memory location or something as complex as calling a sequence of functions. Furthermore, a detection engine is useless without a means through which to enforce the decision. This model is often seen with anti-virus programs today which utilize a decision engine to compare file scanning results to a virus signature database in order to reach a conclusion followed by a kernel driver to enforce quarantines or removal of detected malware.

## **Related Work**

The ultimate goal of our research is to have such a decision-making and enforcement capability available without modification of current programs or major modification to infrastructure and/or software platforms. At the Chinese Academy of Sciences, HyperCrop was developed to perform such actions. HyperCrop utilized the Xen hypervisor in an architecture which spanned userspace to hypervisor-space to enroll applications into the provided protection capabilities[5]. Using this method, an entire operating system would be subject to both the decision engine and enforcement capability since a hypervisor runs at a more privileged level than an operating system running atop it.

In order to achieve their goal, HyperCrop researchers spread out the decision engine in the form of memory-based shadow memory analysis technique leveraging Xen's context switch handler, page fault handler, and debug exception handler while their enforcement engine being a hardware breakpoint to pause a program identified as being under attack. Rather than enroll an entire operating system into the system, HyperCrop's modified context switch handler identifies when the to-be-protected application is the next to be called, then initiates the protection capabilities for the duration of the protected application's runtime. Through their stack analysis techniques, HyperCrop is able to do a software implementation decision/detection engine for application protection from ROP at a hypervisor level.

## **Contribution to Research**

Extending the idea of HyperCrop, the most costly task which they seem to perform is generating the shadow stack and walks through memory in order to find where related components are located. With the upcoming Intel Haswell platform, an additional feature is exposed through the hardware-assisted virtualization capability which provides the capability to inspect within a virtual machine's extended page table (EPT) to see individual guest virtual machine applications. With this, a hypervisor would not require modifications to its components in order to detect when the application enrolled within protection capabilities is about to begin execution. Now, the overhead of searching for relevant sections of memory is eliminated and inspecting a virtual machine's application's virtual memory and mapping it to hypervisor-physical memory addresses is far easier with the assistance of an extended hardware memory management unit (MMU).

To leverage these capabilities, a similar environment to HyperCrop's was installed utilizing a development VT-based modular hypervisor from Intel with Windows 7 32-bit as a guest running applications to be protected. Linking all the components together is a Windows kernel driver and userspace library housing the decision engine. To ensure the decision engine is invoked, we leverage the hypervisor's modularity with a handle from the page fault handler into Windows kernel driver which subsequently invokes the userspace decision engine. Additionally, this architecture would allow for decision/enforcement engine displacement is such a way as to have monitoring virtual machines within a virtual computing environment which are able to pass judgement on the conduct of applications within neighboring virtual machines.

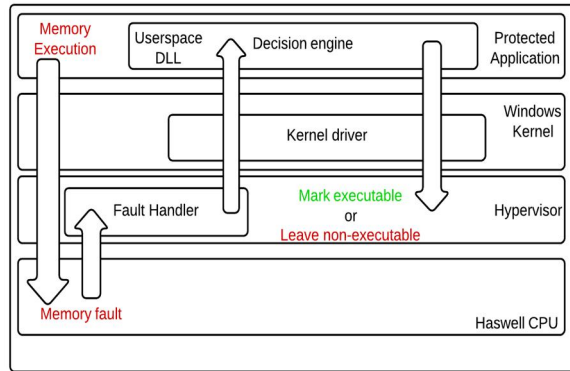


Figure 1. HARD Mode Schematic.

## **Implementation**

When the decision engine DLL is injected into a process, it pauses all other threads in the program. From there, it searches for other modules loaded into the process, and marks them as pages to be watched from the hypervisor. Finally, it resumes all threads. After this, the DLL will only be used as the decision engine whenever invoked from the hypervisor. The decision engine component is comprised of a very simple algorithm which inspects the target of a return instruction and ensures that it points to a location in memory immediately preceded by a call instruction. Based upon the decision made, the engine then either requests a target page marked executable or not. Should the memory access be deemed malicious, we then set the processor instruction pointer to zero - resulting in the operating system terminating the application attempting to execute a nonsensical memory location. This design also provides the capability to replace this decision engine with a more complex decision engine or multiple engines to reach a more reliable and/or correct decision.

## **Test Plan**

HARD Mode's testing is designed to reflect a typical use case for the user applications which often fall victim to ROP. As such, the Intel Haswell testbed reflected a common desktop computer running the Windows 7 32-bit operating system. To make the test seem more typical, the hypervisor is theorized to have come with the computer either from the manufacturer or as part of the HARD Mode security suite. Finally, the user application tested was Adobe Reader version 9.3.4 vulnerable to CVE-2010-2883. This CVE is known as the Adobe CoolType SING Table "uniqueName" Stack Buffer Overflow. The metasploit exploitation framework has the capability to create a malicious PDF exploiting this vulnerability to display a message box.

Additionally, a custom small program was used to ensure HARD Mode can be properly enabled. In order to ensure that this program was spanning page boundaries, we utilized compiler directives within the source code which cause Windows to load components of the program on separate memory pages. During program execution, it calls functions across page boundaries and performs basic printing of messages to console output. At the end of execution, the program then runs in-line assembly which creates a stack which will cause the decision engine to flag the program's execution as unsafe. Utilizing this program, we create a contrived environment to ensure HARD Mode is functioning as designed.

After ensuring HARD Mode was properly working for the test run, said malicious PDF was placed upon the test machine and the Adobe Reader application started. Once the application was fully loaded, the HARD library was be injected into the running process to enroll it into HARD Mode protection capabilities. Finally, the PDF was loaded by the application via the GUI's file opening capabilities. Should the decision



engine be properly invoked pending a page fault and deem one of the malicious PDF's page faults unsafe, HARD should set the Adobe Reader application's instruction pointer to zero, causing the application to crash. This case would be considered a success since the decision engine passed an appropriate judgement and the enforcement engine properly terminated unsafe program execution. Alternatively, if any of these components failed and the malicious PDF caused Adobe Reader to display a message box via the vulnerability, then HARD failed.

## **Results**

Testing demonstrated that legitimate program flows require a great number of special cases in order to properly execute. For instance, there was an issue with the DbgBreakPoint function being mysteriously called, causing program flow to divert into the body of another function via a return operation - causing a false positive. Crossing page boundaries could cause a false positive as well, but was solved by checking if the requested memory location was within the first 16 bytes of a page. Finally, thread creation may have been a problem, but was whitelisted early in testing and not fully explored. Along with having to whitelist various components to test HARD Mode, there were some technical limitations to the implementation which would require a large amount of programming to work around.

Foremost amongst limitations was that libraries loaded into multiple processes could not be monitored, which ruled out protecting system libraries such as kernel32.dll and ntdll.dll. Preventing shared library protection was the design feature that Windows de-duplicates imported libraries to save memory. Since the protection capabilities are employed at the guest-physical memory layer, any changes to the location holding an imported library would affect all processes which had the library mapped in. Furthermore, page faults are known to be extremely slow operations without our additions, so adding HARD Mode only compounded a performance issue with which computer science has been wrestling since the advent of virtual memory within operating systems. Finally, the current library implementation would not support a forking process since it would not spawn a child of itself for each child process of the parent.

The performance impact of page faults combined with HARD Mode detection became apparent during our practical test. After Adobe Reader was executed with the HARD Mode library mapped in, the application's user interface became unusable. The performance impact was so great that the file menu could not even be selected within the interface. The problem seems to stem from long code path which user interfaces take and the exorbitant number of page faults which the code path creates. Even with a null decision engine (allow all page transitions), the application was still unusable. The same problem, however, was not observed with the developmental test application.

The test application simply performed a number of function calls with simple math and little external library use. With HARD enabled on this application, the performance impact was not noticeable. The combination of these results further cements the idea that HARD Mode comes at a larger performance cost when an application crosses more page boundaries and follows longer code paths during execution. From an implementation standpoint, this also could be surmised since the HARD's decision engine is invoked during every page transition.

## **Conclusions and Future Work**

This technology is targeted at applications such as browsers, document readers, and e-mail applications - all of which utilize a user interface. With the current implementation of HARD and the current ratio of processing speed to code path length, this target can not be achieved. Alternatively, statically-compiled binaries or those which use few to no external libraries and are non-forking without graphical user interfaces are possible candidates for HARD protection. Ironically, an application which has these characteristics is intrinsically less vulnerable to a ROP attack and thus would not likely need HARD.

In the future, HARD could take a more specific approach with specific decision engine components designed for specific tasks. For instance, having a small code path which determines whether an attack has occurred within a specific library rather than a general code path which is designed to protect all libraries. Alternatively, HARD could be redesigned with a smaller footprint without having to span user-space, kernel-space, and hypervisor-space. With a faster decision engine, the performance penalty exacerbation of page faults could be lessened. The end condition which any optimizations should accomplish is the shortest time possible to both invoke the decision engine and for the decision engine to reach an intelligent conclusion. HARD Mode's research has provided evidence that page fault-based decision engine invocation is not effective.

With HARD Mode, the goal was to develop a ROP detection mechanism which would protect legacy code without recompilation or binary resynthesis. Utilizing a hypervisor module, Windows kernel module, and user-space library, it was possible to detect some indicators of ROP utilizing a decision engine called upon memory page transitions. Through performance issues and discovering implementation limitations, HARD Mode proved unusable for the target application class. Further work should concentrate on efficiently calling a decision engine which makes extremely fast and intelligent rulings on the state of a protected application. HARD Mode proved promising, but a new implementation model should be considered for additional research.

## **References**

- [1] "Aleph One". Smashing The Stack For Fun And Profit. Phrack, 7(49), November 1996
- [2] "Sinnadurai, Zhao, Wong". Transparent runtime shadow stack: Protection against malicious return address modifications. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702>, 2008.
- [3] "Shacham, Page, et al.". On the Effectiveness of Address-Space Randomization. In ACM Computer and Communication Security Symposium, 2004.
- [4] "Kc, Keromytis, Prevelakis". Countering code-injection attacks with instruction-set randomization. In Proceedings of the 10th ACM conference on Computer and Communication Security, 2003.
- [5] "Jiang, Jia, et al.". HyperCrop: a hypervisor-based countermeasure for return oriented programming. In Proceedings of 13th International Conference, ICICS, 2011.