COFI Break

Breaking Exploits with Practical Control Flow Integrity

Shlomi Oberman

Ron Shina

Abstract

Software exploitation is an ongoing problem that industry players and academia have been trying to solve for over 2 decades. Despite many attempts and massive resources directed at stopping software exploitation, it is still prevalent and may be a bigger problem in the past several years than it has ever been. Adding to that, the growing reliance on technology in our society multiplies the dangers and reach of software exploitation.

In recent years the use of anti-exploitation and other defensive measures has become commonplace and a basic practice. At the same time awareness of users and developers to these problems have risen. With many defenses in place, exploits are still a weapon of choice for compromising computerized systems and the exploitation landscape is always evolving. In this paper we examine a novel approach to for detection and later prevention of memory-corruption exploits without prior knowledge of the exploit itself in a way that is practical, fast and very challenging for attackers to bypass. We use CPU-based tracing and CFI to do this and show that a practical CFI engine can be built using this method.

Background - CFI

CFI is a topic that has been widely discussed in recent years, mostly in academic circles, with some commercial implementations. In most cases real-world CFI implementations rely on code insertion during compilation with a dedicated build chain or after compilation with DBI and lately also with the use of the processor PMU (Performance Monitoring Unit). The latter is very new and not yet well understood and the first two approaches have their drawbacks – namely the adoption rate and performance overhead of compile-time instrumentation, the performance overhead of DBI passed approaches and the compatibility challenge encountered with both approaches. Some advancements such as Microsoft's CFG and Intel's CET look promising in that they are robust from an engineering perspective and are being pushed for wide adoption by major vendors. In the past 2 years we have also seen some talk of tracing mechanisms for exploit detection, but to this date these are closed-source endeavors.

Our Research and project furthers existing knowledge on this subject by introducing a new approach for instrumentation that is - in our opinions - very powerful and more practical and effective then existing methods. We are also releasing much information about this method so that it can be used and built upon in the future.

Our research into a practical CFI implementation was directed by the framework of being implemented in some sort of gateway appliance or software where it can scan content going in and out of an enterprise network or other computerized environment. Some guidelines result from this framework which we feel make the approach more practical for real-world implementations, and better as a basis for CFI:

- Must work on windows
- The analysis must be fast and robust
- The entire control flow must be analyzed
- The solution can't be easily bypassed by an attacker
- The complete system must be non-intrusive and undetectable by an attacker
- Many types and implementations of CFI should be able to run on top of this implementation

Background – Intel Processor Trace

Intel Processor trace is a wonderful feature in Intel CPU's. It allows low-overhead tracing of every single branch and effectively every single instruction a CPU executes opening a vast world of options.

It is a new-ish feature which was introduced in 5th Gen Intel CPU's (formerly Broadwell). If you read the documentation carefully you might conclude that the feature existed in previous generations and went undocumented. The authors have no knowledge if this is the case.

Intel processor Trace is the youngest sibling in a family of trace mechanisms introduced by Intel over the years and is meant to be used by developers and software architects for performance –tuning and analysis purposes. We found this feature to be a prime candidate as an instrumentation method for CFI purposes because, well, it is meant to be a fast and practical instrumentation mechanism and it has proved to be exactly that.

Approach

This project utilizes Intel processor trace (IPT) in order to detect deviations from control flow integrity and enforce CFI. In simple terms – we cause a program to parse a known file format and trace the flow of the program, after this we analyze the trace to detect any CFI deviations. We are in-fact utilizing Intel PT in fashion for which it was not originally intended, applying it as an off-label solution. This paper examines the details of using IPT as a platform for applying CFI of any type thus detecting memory-corruption exploits in any type of content in a practical manner on a windows platform. In order to achieve this we must adhere to certain requirements – namely, full visibility of the complete program flow and a certain level of performance. In order to fulfill these requirements we base the solution in its entirety on the use of Intel PT and trace the entire program flow (vs. partial tracing).

Tracing the complete flow of a program using IPT requires use of the IPT trace and a memory map of the process in addition to continuous disassembly. This is computationally expensive and thus a large part of the effort require to develop this project was invested in improving performance to a point at which it is

no longer a barrier. This means several seconds analysis time per file for common file formats, with a clear path to further reducing this time.

In order to demonstrate the CFI capabilities of this approach we implemented a shadow-stack verifying the return address of every function and matching it to the caller address with minimal performance overhead. In a similar manner any type of forward-edge or other CFI can be implemented with minimal overhead and full visibility to the program flow.

As a side-effect of exploit detection this approach also allows for exact pinpointing of a large part of the exploitation process. With the assistance of the shadow stack – as an example - we are able to immediately see the entire ROP chain of an exploit that uses ROP. This information could be very valuable for analysis of malicious content during incident response or many other scenarios and could lead to faster resolution of alerts produced by a solution using this method.

There seems to be an inherent trade-off between performance and security of early implementations of hardware-assisted solutions, where PMU or statistically-based solutions tend to have better performance at the expense of security and the precision of the CFI. It is our opinion that a completely IPT-based solution can be fast enough to operate as a prevention (vs. detection) solution and also that careful analysis will show a delicate mix of PMU and Trace-based approaches can achieve the best of both world.

Design

To achieve high performance, full visibility and precise flow reconstruction some design and implementation details must be considered.

Thread Tracking

Intel PT trace information does not include any OS-specific or OS-related data other than allowing filtration of trace operation according to the value of CR3. Therefore in order to reconstruct the flow of a program we must have some outside source of thread context for every part of the trace. Several ways to reconstruct thread context were considered. Among these are:

- Thread context can be extracted by scheduling APC routines on every thread and detecting the calling of this routing in the trace data (one routine per thread). This would potentially somewhat reduce the non-intrusiveness attribute of the solution and is still considered a good option.

- Reconstructing thread context based only on the thread data and the address at which the trace stopped and started together with the thread call chain, allowing for ambiguous states where a trace may be one of several threads during a certain point in time and “locked-in” as soon as a unique return address is encountered. This was abandoned due to a resulting weaker CFI model and the fact that in real windows applications many threads are waiting on objects and have a similar call chain at any given moment).

- Use of ETW (event tracing for windows) can be used to track thread switches and match them up to timestamps in the trace. This failed due to insufficient granularity and preciseness in the IPT timestamps. This may have improved with 6th gen CPUs.

- Use of kernel detours on the context switch routines. This was our chosen solution and requires a bypass of Microsoft Patch Guard, for which there are some off-the-shelf solutions.

- In a virtualized environment new options are available for exploration such as the use SLAT-based memory breakpoints on context switching functions and reading of the guest memory to discover the context. Other solutions may exist in virtualized environments (where the host is accessible of course).

Module tracking

Analysis of Intel PT trace information requires an exact mapping of memory at any given moment. In modern OS environments and specifically windows this means tracking loading and unloading of dynamic modules. This was performed by using ETW to get dynamic module loading order and then synchronizing the load order with the trace at certain points where the control flow reaches a function that causes loading of a module (for example: NtloadLibrary). Module unloading is synchronized lazily, meaning that whenever a module is loaded, every unload that occurred prior to this load is performed. In some cases a program may load a module "manually" by allocating memory and performing parsing of the PE format or similar methods. We encountered this in some cases. This can be solved by adding more synch points such as section mapping or file reading or adding some outside information similarly to the method used with thread synchronization. Fortunately these cases are quite rare and very specific and solving them on a case-by-case base is sufficient.

Multiple processor trace coalescing

A single process consisting of several threads may execute simultaneously or interchangeably on multiple processor cores. For this reason we would have to stitch together traces from several different cores. We decided to side-step this issue by tracing on a single-core and forcing the process to execute on the same process by use of the processor affinity windows API. In general this approach is also optimized for analysis of multiple files simultaneously on one hand and may result in an attacker detecting the system on the other hand (by comparing timing of a multithreaded vs. single-thread operation). Further effort is required to produce a trace consisting of several processor cores simultaneously. To the best of our knowledge an open-source implementation of a windows-based PT trace driver is scheduled to be released during 2016.

Process orchestration

We need an orchestration method in order to start the traced application and "pass" it the traced file. It would be preferable that this method be robust and also allow us to trace as little as needed and be expandable In the sense that it allows us to perform actions with the content if this is requires in the future (move to a certain page in a pdf etc..). To achieve this there are several orchestration implementations using OLE automation for Microsoft office, Dynamic Data exchange for adobe acrobat, windows messaging and commandline for further expansion and other programs. These will not be described in detail as they are not the main focus of this whitepaper.

Optimization

A naïve approach to analysis is achieved by taking a memory map, an IPT trace and a disassembler and using them in tandem to analyze the program flow. Using this method to analyze an IPT trace given a complete trace of common programs parsing common file formats used in enterprise environments (for example: ppt on Microsoft powerpoint) would require many hours of analysis. This is obviously impractical. A quick analysis will show that much of the analysis performed is superfluous, since we are disassembling many functions that do not have an immediate effect on the flow of the program (such as MOV, ADD etc..)

Optimization Implementation Details

Pre-analysis

We developed an optimization technique consisting of pre-analysis of known binaries such as windows dlls, and the dlls and executable files of the programs we will be tracing. Using IDA and IDAPython we built a graph of the basic blocks in each module. We can then use the graph in conjunction with the IPT trace data instead of performing disassembly of every instruction, mapping information from the trace to nodes in the graph and traversing the graph according to the trace.

Fast Graph lookup

It becomes apparent that as soon as we consider using a pre-calculated graph of basic blocks in every model that we will also be adding a “step” of looking up the correct node in the graph after very control transfer (moving between BB with jmp, call, ret etc..). We must therefore consider how to minimize the lookup time. Arranging this graph so that the graph information is actually stored in the same relative location as the Basic blocks themselves allows us fast lookup times since the branch offsets give us the destination BB immediately. If we further arrange this data in a PE format we’ve essentially used a pseudo-assembly-language that reduces x86-64 to control flow data only and is much faster to disassemble. Using this method allows us to reduce analysis time from hours to seconds-minutes and is a vast improvement. We call this method “shadowing” since we create a shadow of the original executable or library file with redacted information about the code contained within. We have never attempted to use a graph without this optimized lookup, it stands to reason that results would vary widely depending on the lookup function. In this project the BBs are arranged according to their relative addresses in every module and are stored in a PE format, however the different pseudo-modules containing the basic blocks are not loaded to memory in relative addresses equal to those in the original program and so a lookup must be made on every module switch leading to some performance overhead. Future work may reduce this requirement and place all BBs in accurate relative positions and/or in their actual location in memory which will further reduce the need to calculate the next BB address by subtraction. Hints of which module must be fetched to find the next BB may also be incorporated in to the shadowed data (for example: when using imports).

Backup disassembly and context-switching

Static code analysis is a tough challenge under any real-word circumstances and even taking a binary module and extracting the basic blocks in the code doesn't always result in a perfect result when object oriented programming or functions pointers are used. Basing analysis on IDA scripting also has some limitation but is much more practical then writing an x86-64 binary analysis platform from scratch. Moreover, developers of applications will sometimes seek to use anti-reversing or anti-debugging tricks that elude the standard representation of a BB-graph. This means that in some cases during analysis we will find ourselves branching to a location in memory that is not the beginning of a basic block or for which we do not have basic block information. This can also happen in the case that a context switch (thread start-stop) doesn't occur at the beginning of a basic-block. To overcome these issues we use a backup disassembly module as well as a "module map". The module map is stored in a PE format and contains a marker for each byte in the module. This marker represents if this area is analyzed as code and - if it is indeed analyzed as code - how far that byte is from the start of the BB. This map is then consulted to determine whether we should look several bytes back in memory for the start of the BB or default to disassembly using a backup module stored in memory which contains the original bytes of the PE file. An observational reader will notice that in some cases we must use the disassembly method even if we find that we are analyzing an area which is marked as code. For every BB we should be able to know what analysis method to use (BB, disassembly or back-step.)

Code-patching

As specified above, in some cases (as well as some cases not yet mentioned) we will need to perform disassembly during analysis. This is a time-consuming endeavor and takes up a large portion of the analysis time in empirical testing. Many of these assembly instructions will be disassembled over and over as they are executed many times. In some of the cases where we need to perform disassembly we can perform an on-the-fly shadowing of the code, or as was implemented in practice – patching of the disassembly code to remove all superfluous instructions which do not cause control transfer. This essentially means putting the last opcode of the BB in place of the first opcode. Careful readers will notice that this will not always work in cases of anti-reversing tricks or if the BB is very small and does not end in a control transfer instruction. This is not an issue since we only perform this operation on larger basic blocks for performance reasons (it's not cost-effective for small BBs which end in a branch). This could still pose a problem since we don't know the real size of the basic block if it does not end with a branch but is not an issue due to the nature of real-world code and some specific implementation details. We could also avoid patching for certain modules or for certain code patterns, however in empirical testing this was not necessary.

Partial disassembly

It would make sense to attempt performing disassembly to only extract needed information from every opcode. That is – complete disassembly of control transfer opcodes and the extraction of length only from other opcodes (as well as which type of opcode is being analyzed). This is somewhat tricky to perform optimally due to the nature of the x86-64 architecture which often requires analysis of the exact opcode and the type of operands and to get the complete opcode length, however some progress and optimization can definitely be performed. This type of optimization was not performed during the

described project, however it is the assumption of the authors that this type of analysis is possible and has been likely been performed by different individuals, teams or companies for the purposes of tracing for performance measurement or other purposes.

Other implementation challenges

Dynamic code generation

Dynamically generated code produced by JIT compilers is a part of many modern applications. Handling this type of code was not in the scope of what was developed except for some minor edge cases which were handled in order to support the Microsoft office WARP JIT engine. We can only recommend a path to supporting dynamically generated code. We can recommend several options, of which a combination will likely be most beneficial:

1. It is clear that a snapshot of the code will need to be taken in order to analyze it and that this snapshot will need to be taken every time a page is executed. This can be performed by piggybacking on existing W^X policies in existing applications or by using memory protection on JIT pages (optionally using SLAT) as a breakpoint mechanism to enforce W^X and then take a snapshot upon execution. This is slightly similar to what was performed in CXInspector (Ralf hund. et al)

2. Another path would be to take snapshot of the CPU and thread state before JIT code emission which will allow partial emulation of the program from this point to make sure where JIT code is being emitted from. This might prove reasonable from a performance perspective.

3. A third option is tracking the flow in and out of JIT code looking at the side effects of the code. For example – if a function calls in to JIT code or returns in to JIT code in an unexpected manner or if JIT code causes the execution of specific system calls.

4. A fourth option would be to use PMU events, registers or counters to deduce information about code execution in the JIT. These can be LBR, branch mis-predictions, stack trace on supported processors or other features.

Errata

Like any new HW feature, IPT has some documented and undocumented flaws. These usually result in fairly deterministic bad trace data that does not conform to what actually happened during the program execution. This is handled on a case-by-case basis, performing a fixup every time such a state is detected. Because of the nature of IPT analysis usually a bad trace will eventually result in a mismatch in the number of TNT bits seen vs expected in the trace. In some cases data may need to be pulled from the program binary and in some cases a fixup of the current analysis state must be performed. In some very improbable cases this could lead to weaker CFI and to an attacker leveraging some of these errata to manipulate an analysis engine. This is a currently a very theoretical attack.

Debugging

Since analysis runs over hundreds of thousands of iterations of an analysis loop during short runs, and many more during long runs it becomes challenging to debug the program and find problems in the analysis code. This was solved using carefully placed debug assertions and prints and utilities allowing an initial breakpoint to be triggered at a certain “position” in the trace file using a command line argument.

“Unnatural” flow

An IPT analysis engine that reconstructs program flow requires a shadow stack to handle the “return compression” feature of IPT which emits a single bit instead of an address to signal that a functions returns to the address following the address from where it was called (as it should in most cases). Furthermore, a shadow stack helps identify code-reuse attacks and is a part of many fine-grained CFI models. Unfortunately, in the windows operating system and in several specific programs some mechanisms work by changing the “natural” behavior of functions returning to their caller. This might cause the analysis to trigger a code-reuse alert which is false-positive. To handle this we analyzed every mechanism where this happens and added code that handles these on a case by case basis. This issue also requires hardening of allowed control flow since these areas are a soft underbelly for attackers that wish to cause functions to return elsewhere and make it look like a legitimate part of the program.

Results

We run the analysis on a corpus of approximately 10,000 benign PDFS as well as several thousand ppt and docx files and found that the analysis time per file is as low as 10seconds and no higher than 2 minutes in rare cases. The files were parsed with adobe reader 9.34 and adobe reader DC for pdf, and with office2013 for office documents. The detection did not report any false-positives for this data set. It is worth mentioning that the corpus included long graphic-laden PPT files meant for child education and that it is our opinion that real world samples in a corporate environment will not take more than 30 seconds to analyze. All the analysis times exclude the time it took the program to parse the file and were performed on a single core of a 5th generation i7-5500u CPU on a mid-range laptop. We also analyzed several malicious files produced from metasploit (and derivations thereof) which were reliable and used code-reuse. The analysis engine detected all of the malicious files as malicious without receiving any prior knowledge of the exploit and showed the address of the first gadget of the ROP chain which was not known to us or the analysis engine prior to analysis.

Counter-measures

The work presented above is –in the author’s opinions – one of the most practical ways to perform CFI on binary programs post-compilation. Working in tandem with a string CFI model based on sound binary-analysis and existing anti-exploitation this system can detect and stop today’s most advanced exploitation techniques. Having said that, no security solution is a silver bullet. Attackers are constantly sharpening their tools. The way to bypass this solution would depend very much on what type of CFI restrictions are placed on the program flow as well as the details of the attacked program.

It has been shown lately that exploitation can occur without direct control flow manipulation by use of data modification attacks. These attacks however, often rely on a specific CFI model and a specific implementation to work. Given a universal platform for analyzing flow some of these attacks might not be feasible as they can be detected either because they are very specific in nature (such as use of the print formatting family of functions) or because they will be detected earlier in the exploitation. For example: a UAF turned in to an object-forging attack could be detected by a model that knows to restrict virtual call target addresses according to type information or prior dynamic or static analysis. Still, an attacker able to reach a universal write-what-where vulnerability in conjunction with an information leak without being detected is likely to be able to carry out the exploit all the way to payload stage.

And of course, some exploits do not involve memory-corruption and so are out of the realm of this and many other exploit detection and prevention systems.

Conclusion

Throughout the history of information security, and despite the development of many anti-exploitation techniques, exploit developers were always able to outsmart defenses and produce viable working exploits. This status is not any different today, existing solutions slow attackers down and raise the bar but do not completely stop exploitation. New methods are needed if we are to keep up with attackers. In this paper we demonstrated a novel, practical and effective platform for fine-grained CFI implementations. This approach improves upon compiler-based methods and DBI based methods in many aspects and is different than PMU-based methods, having an apparent performance-efficacy tradeoff with early implementations of the PMU approach. Using this method it is truly possible to enforce or detect deviation from any CFI model, subject only to performance constraints of the CFI model itself. Thus tracing becomes a platform for CFI.

The method advanced the current state of the art and is a stepping stone for more effective exploit detection and prevention. Using statically or dynamically generated CFI models with this platform can raise the bar substantially for attackers, making exploitation even harder than it is today. We believe that there is much future work to be done in the field - either in the implementations of CFI models which can be used with this platform on specific programs, in future performance improvements to the method or in novel combinations of tracing and PMU events.