Technical Analysis of the Top BlueHat Prize Submissions

swiat 26 Jul 2012 9:55 PM

Now that we have announced the winners of the first BlueHat Prize competition, we wanted to provide some technical details on the top entries and explain how we evaluated their submissions. Speaking on behalf of the judges, it was great to see people thinking creatively about defensive solutions to important security problems!

To set the stage for this post, we thought it would be helpful to quickly remind everyone of the problem that entrants needed to solve. Specifically, entrants were required to design a novel runtime mitigation technology that would be capable of preventing the exploitation of memory safety vulnerabilities (such as buffer overruns). The BlueHat Prize judging panel was then responsible for evaluating each submission as described according to the following criteria (as described in the contest rules):

1. Practical and functional (30%)
    a. Can the solution be deployed at large scale?
    b. Does the prototype have low overhead?
    c. Is the prototype free of any application compatibility or usability regressions?
    d. Does the prototype function as intended?
2. Robustness (30%)
    a. How easy would it be to bypass the proposed solution?
3. Impact (40%)
    a. Does the solution strongly address key open problems or significantly refine an existing approach?
    b. Would the solution strongly mitigate exploits above and beyond Microsoft's current arsenal?

The judges for this contest consisted of representatives from Windows, Microsoft Research, and Microsoft's Security Engineering Center (MSEC). Of the 20 entries received, the top three submissions described different methods of mitigating return oriented programming (ROP). Let's dive into the technical details of these submissions.

3rd place: mitigating ROP via return site whitelisting (/ROP)

This entry, as submitted by Jared DeMott, described a method of imposing a whitelist on the set of locations that a return instruction can transfer control to. More specifically, this solution calls for the introduction of a new compiler flag ("/ROP") that would add metadata to an executable that describes the set of valid return sites in the image. When the image is loaded at runtime by the operating system, the image's list is added to a master list. As the program executes, each invocation of a return instruction triggers an exception that causes the operating system to validate the target return site against the master list of return sites. If the target return site is in the list, the program continues executing as normal; otherwise, the program is safely terminated.

To prototype this idea, the submission included a Pin tool that simulated the hardware support that would be needed to augment the behavior of the return instruction. In addition, the prototype also included an IDA Python script to identify the set of valid return sites for an image. This script generated the input that was needed for the Pin tool to check whether a return target was to be considered valid.

Practical and functional

Although this solution is functional, it is not seen as practical for large scale deployment as described. The primary reason for this is due to the execution cost associated with implementing this check. This cost is expected to be significant because the design calls for a software interrupt to be raised for each return instruction. A second issue with this design is related to the data structure that is used to store the address of valid return sites. In particular, the prototype of this design uses the STL map container which, although it enables O(1) lookups, is not optimally compact and can therefore lead to considerable memory overhead depending on the number valid return sites that exist in modules loaded by a process. The design for this solution did not propose optimizations that could help to address both of these concerns.

Robustness

This solution is seen as a partial mitigation for ROP. It could be bypassed by leveraging gadgets that are in the set of valid return sites or by using a gadget chaining method that does not involve a return instruction. The feasibility of finding a sufficient set of gadgets in the set of valid return sites is expected to be uncommon. The use of alternative chaining methods is feasible, although the complexity associated with doing so exceeds the current state of the art in ROP-based exploits seen in the wild.

Impact

This solution would have a moderate impact if were possible to deploy it at large scale. The fact that this solution does not fully address all forms of code re-use limits the expected long term impact of the design as described.

2nd place: mitigating ROP by placing new checks in critical functions (ROPGuard)

This entry, as submitted by Ivan Fractic, described a method of mitigating ROP by introducing additional checks that are performed when critical functions, such as VirtualProtect, are called. These checks are designed to detect conditions that are indicative of ROP occurring, such as an API being called out of context. The checks proposed by this submission included:

1. Verifying that the stack pointer is within the bounds of the thread's stack.
2. Verifying that that the return address of a critical function is executable and preceded by a call.
3. Verifying that all stack frames are valid and satisfy criteria 1 and 2.
4. Simulate execution forward from a critical function's return address to verify that subsequent returns satisfy criteria #2.
5. Function specific contract changes (e.g. prevent reprotecting of the stack as executable).

Although adding checks to critical functions to detect ROP is not a new idea, and indeed some of the checks above have already been described in previous research, this submission included novel elements that we had not seen discussed before. We actually received a number of submissions which proposed adding new checks to critical functions, but the other submissions had a subset of the checks proposed by this submission or by previous research.

Practical and functional

The checks proposed by this submission are considered to be both practical and functional. By limiting these checks to certain critical functions, the performance impact is minimized. Some of the proposed checks would be incompatible with certain applications. Specifically, criteria #1 is known to be incompatible with some legacy applications that do custom stack switching and #3 is incompatible with x86 programs that enable frame pointer omission.

Robustness

ROP mitigations that rely on introducing new checks to critical functions are not considered to be robust over the long term. The checks proposed by this submission and in previous research are capable of mitigating ROP payloads that are used today, but it is expected that attackers would be able to adapt to these checks at relatively low cost. For example, a fundamental problem with this type of approach is that an attacker could attempt to call a lower level API that has not been instrumented by the checks. A variant of this bypass is to transfer control after the instruction block that performs the checks (depending on how the checks have been added).

Impact

This solution would have a moderate impact if implemented correctly and deployed at large scale. The fact that this solution does not fundamentally address ROP limits the expected long term impact of the design as described.

1st place: mitigating ROP via Last Branch Recording (kBouncer)

This entry, as submitted by Vasilis Pappas, described a novel method of using the Last Branch Recording (LBR) feature of Intel processors to detect ROP when system calls are made. This method relies on a kernel component that allows branch recording to be enabled for return control transfers. When a system call occurs, the kernel component then enumerates each entry in the LBR stack and verifies that the destination address is preceded by a call instruction. The prototype for this solution relied on evaluating the contents of the LBR when certain critical APIs were called rather than at the system call layer. The cited reason for this was due to Windows kernel restrictions around interposing the system call layer in kernel mode.

Practical and functional

This solution is considered to be both practical and functional. The use of supported hardware features to track the destination address of return control transfers helps to drive down the performance cost and complexity of implementing this solution. There should also be minimal application compatibility impact through this approach.

Robustness

This solution is not expected to be robust over the long term, although it should be robust against ROP payloads that are used today. There are multiple reasons why it is not expected to be robust. First and foremost, the Last Branch Recording feature of Intel processors has a limited stack for storing control transfers (16 entries on Nahelam and up). If an attacker can ensure that a sufficient number of valid returns happen between the control transfer that eventually leads to an API call and the point where the LBR stack is checked, then this solution can be bypassed. It is believed that attackers would be able to accomplish this in most cases with a low to moderate development cost. While it may be possible to use the Branch Trace Store (BTS) feature to help address this problem, the performance cost may become unacceptable. The other reasons for this solution not being robust are shared with the two previous submissions: specifically, imposing these checks on specific APIs (as in the prototype) may be prone to bypasses and imposing checks only on returns does not mitigate all methods of chaining gadgets.

Impact

This solution would have a moderate impact if implemented correctly and deployed at large scale. The fact that this solution does not fundamentally address ROP limits the expected long term impact of the design as described.

Closing thoughts

The winning submissions illustrate some of the creative thinking that has gone into developing defensive methods of making it more difficult and costly to exploit memory safety vulnerabilities. As we look toward the future, we will investigate whether there are elements of these methods that may make sense to integrate into EMET or a future version of our products. We have already taken steps in this direction by integrating features from the ROPGuard submission and related prior research into the Technical Preview of EMET 3.5.

As the judging criteria makes it clear, it can be quite challenging to turn an interesting defensive security idea into something that can be shipped in a retail product at large scale. Nevertheless, ideas that may seem impractical on the surface can eventually be turned into an innovative and practical solution – it just takes some additional focused thinking.

Matt Miller

MSEC Security Science