Multi-Variant Program Execution for Vulnerability Detection and Analysis*

Todd Jackson Christian Wimmer Michael Franz
Department of Computer Science
University of California, Irvine
{tmjackso, cwimmer, franz}@uci.edu

Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: General — *Security and protection*

General Terms

Design, Security

Keywords

process monitoring, vulnerability detection, multi-variant execution

1. INTRODUCTION

Software vulnerabilities continue to be a major threat. Although significant advances have been made to reduce such vulnerabilities, there are still vulnerabilities that have eluded these techniques, and unfortunately the attackers have also become more sophisticated, employing more devious methods. Moreover, a huge amount of new code is written every year, so that even though the *error rate* may be decreasing, the *overall number of vulnerabilities* is still increasing. For example, the number of buffer errors listed in the National Vulnerabilities Database increased from 398 in 2007 to 563 in the year 2008 [7].

Multi-variant code execution is a run-time technique that prevents the execution of malicious code. It does not remove the vulnerability underlying an attack, but it prevents the vulnerability from being exploited by an attacker. The key idea is to run two or more slightly different variants of the same program in lockstep on a multiprocessor. At certain synchronization points, their behavior is compared against each other. Divergence among the behavior of the variants is an indication of an anomaly in the system and raises an alarm.

Similarly, work on determining the exact locations of errors in software is still ongoing. While there is progress in software healing and hot patching, software vendors often find out about errors when a proof-of-concept exploit is released. Multi-variant execution allows for the creation of a system that is an effective honeypot. This allows for the direct comparison of state between variants, reducing the time needed to create and deploy a patch. This lets security researchers create honeypots that are not limited by the ability to emulate known vulnerabilities, and thus creating a significantly more realistic honeypot.

In this paper, we provide a brief overview of multi-variant execution, summarize some variation techniques, illustrate a honeypot system constructed from a multi-variant execution environment, and then list combinations of variation techniques that are effective in multi-variant execution environments.

*This work is partly funded by the Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSIIRW '10, April 21–23, 2010, Oak Ridge, Tennessee, USA.
Copyright 2010 ACM 978-1-4503-0017-9 ...$10.00.

2. MULTI-VARIANT EXECUTION

Multi-variant code execution is a run-time technique that prevents malicious code execution by running a few slightly different instances of one program in lockstep and comparing their behavior against each other. These instances are called *variants*. A monitor is responsible for synchronizing the variants and comparing their behavior at certain points during program execution. Divergence among behavior of the variants at these synchronization points is an indication of an anomaly in the system and raises an alarm.

A Multi-Variant Execution Environment (MVEE) duplicates the in-specification behavior of a program without duplicating the vulnerabilities to which the variants are immune. This characteristic allows effective monitoring systems that can detect exploitation of vulnerabilities at run time before the attacker has the opportunity to compromise the system. In a MVEE, input to the system is simultaneously fed to all variants. This design makes it extremely difficult for an attacker to send individual malicious input to different variants and compromise them one at a time. If the variants are chosen properly, a malicious input to one variant leads to collateral damage in at least one of the other variants, causing a divergence. The divergence is then detected by the monitor.

Multi-variant execution is a monitoring mechanism that controls states of the variants being executed and verifies that the variants are complying to the defined rules. A monitoring agent, or *monitor*, is responsible for performing the checks and ensuring that no program instance has been corrupted. Userspace monitors isolate the processes executing the variants from the OS kernel and monitors all communication between the variants and the kernel. At the same

time, the monitor is a separate process with its own address space. Therefore, it is difficult to compromise the monitor by taking control of a variant. Kernel-based monitors perform a series of checks before allowing the rest of the kernel to operate on behalf of a variant.

The monitor can operate at different granularities that do not change the types of attacks that can be detected, but it is a factor in determining how quickly attacks are discovered. For example, Orchestra [10] uses system calls as synchronization points, preventing damage to systems such as erasing files or network communication. A more fine-grained monitoring level is at the instruction level, where hardware assists in the synchronization effort. Before each instruction is executed, the instructions are examined to ensure that the instructions and operands are the equivalent. This prevents payload code from executing. However, scheduling of multi-process or multi-threaded applications, asynchronous signal delivery, time, random numbers, file and process identifiers are major sources of inconsistency among the variants. Those interested in techniques to remove the sources of inconsistency are referred to [10].

3. VARIATION TECHNIQUES

In this section we discuss several behavioral variations that can be applied to an executable. All variants can be generated at compile time, and some of them can be performed by manipulating binary executables. The list contains some well understood variation techniques, but we also introduce new variations that are simple and may appear to be ineffective, but are powerful in the context of a multi-variant execution environment. We describe only approaches that do not change the internal behavior of an executable in such a way that run-time comparison with other variants becomes impossible.

Reverse Stack Growth Direction. Most processor architectures are designed for one stack growth direction, but by augmenting the stack manipulation instructions with additions and subtractions of the stack pointer, it is possible to generate a variant where the stack grows in the opposite direction. This defends against historic buffer overflows and classic stack smashing attacks that rely on a particular stack growth direction because the stack layout, including buffers and variables allocated on the stack, is completely different.

Instruction Set Randomization. Machine instructions usually consist of an opcode followed by zero or more arguments. The processor uses the opcode to determine the type of instruction to execute. Therefore, randomizing the opcodes creates a new instruction set. This can be done by simply XORing the opcode with a known value. If an attacker injects code that is not properly encoded, it will still go through the decoding process just before execution. This leads to illegal or incorrect code that does not perform as intended because the processor will not decode the attacker's code properly.

System Call Number Randomization. All payloads that embed system calls have to know the correct system call numbers. By changing the system call numbers, the injected code executes a system call that leads to completely different behavior or even an error. One disadvantage is that either the kernel or an agent must reverse the randomization process before the system call is executed.

Library Entry Point Randomization. Another possibility to gain control over a system is to call directly into a library instead of using hard coded system calls. Guessing the addresses of the library functions is fairly easy since similar operating systems tend to map shared libraries to the same virtual address. This can be defended by randomizing library entry points and is implemented by rewriting the function names in the binary or during load time.

Register Randomization. Register randomization exchanges the meaning of two registers. Many attacks assume that registers have a particular use. When attack payloads that depend on these assumptions are run on systems where the assumptions are not valid, the payload will not behave as expected.

4. MULTI-VARIANT EXECUTION IN HONEYPOTS

Multi-variant execution environments, when configured for system call granularity, will raise an alarm when detecting that one or more of the variants has diverged from the others. However, at this point, it is impossible to find the root causes of the divergence, since the damage has been done. This is a significant problem for those trying to create patches for security-related bugs based on a proof-of-concept exploit, since they convey little information about the state of the vulnerable program when it is exploited. On Unix-based systems, core dumps provide a dump of the target process' state, but this is done only if the target program crashed and was not exploited successfully.

Honeypots such as *nepenthes* [2] and its successor *dionaea* [1] require knowledge of an exploit in order to emulate the behavior of vulnerable software. This emulation is used to trick an attacker into delivering its payload. When confronted with a previously unknown attack, the honeypot will emulate the behavior as much as possible. Its ability depends on the amount of emulation that is implemented in the honeypot software. As a result, honeypots like these are effective at showing how known exploits are used, and are good at sampling the software used to create botnets and their propagation. However, they are unable to detect zero-day and undisclosed exploits because the honeypot does not share all of the characteristics of the target software.

Multi-variant execution environments bridge the gap between the safety of a honeypot and the risks of running Internet-facing software. This is due to the fact that multi-variant execution environments are capable of running fully capable software while protecting the host system from damage caused by arbitrary code execution attacks. As a result, because the monitor has stopped the variants due to attack, a honeypot monitor performs a detailed analysis of the variants to determine how they were affected.

In order for the monitor to be able to perform this kind of analysis, the monitor needs to take a few extra steps before and during execution of the variants. First, before execution, the monitor needs to be told which variation techniques were used on the variants. This is required knowledge for the monitor in the case of system call number randomization as the monitor must swap out the randomized system call number with the proper number as the operating system kernel is variant-agnostic. However, in cases like heap layout randomization or reverse stack growth, the monitor needs to know that the memory maps of the variants cannot be directly compared.

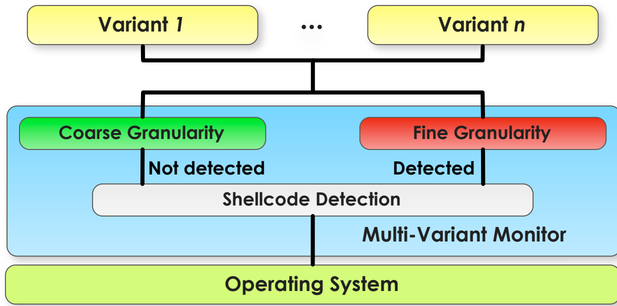

Figure 1: When the monitor's shellcode detector determines that there is data designed to compromise a variant, the monitor switches to a finer grained monitoring level.

Secondly, on execution of a system call on behalf of a variant such as `read`, the monitor checks for shellcode in various forms. A critical part of performing an arbitrary code execution attack is the careful crafting of an input that corrupts the variant, takes control of it, and then executes code delivered by the attacker. This presumes that the variant must accept the attacker's input at some point. Well known shellcodes can be found with signatures and string matching engines. More sophisticated payloads utilize *polymorphic code*. Polymorphic code can be detected using heuristics.

Third, on detection of a payload containing shellcode, the monitor switches to a fine-grained form of monitoring. Hoisting the executable on to a system which instruments binaries such as one provided by Valgrind [8] or Strata [11] allow the monitor to examine what the process is doing at a lower level. This finer-grained monitoring permits tracking the processes across functions and instructions instead of between system calls, which is possible because the variants are created from the same source code. Recording an execution trace from the system call where attacks were introduced to the time of system call divergence gives a short, yet detailed explanation of how attackers are exploiting vulnerabilities. Comparing against other variants shows the data structures that were corrupted in order for the attacker to take control of the variant.

In case the detection heuristics or signatures do not detect shellcode, the monitor still performs some post-mortem analysis. For example, the monitor has the option of inducing a core dump of both variants and comparing the memory maps. The monitor also compares the program stacks to determine where in the variant's code the attack was attempted.

5. CHOOSING OPTIMAL VARIATIONS

Determining the optimal set of variants to be used in a multi-variant execution environment is done by finding the set of variants which provides the maximum combined coverage. Because of the overlap in the level of protection that the different variation methods provide, we define the *optimal variants* of an n-variant MVEE to be the variants whose combined coverage is maximized over all combinations of n available variants.

For two-variant MVEEs used purely in a context where detection is the priority, the best combination is to choose one of instruction set randomization, system call number randomization, and register randomization and use that in concert with library entry point randomization. These variation techniques are very effective at rendering shellcode inert. This combination is sufficient to cover the arbitrary code execution vulnerabilities listed for Apache, MySQL, and ISC BIND from 2006 to 2008 in the National Vulnerabilities Database, as well as the majority of the Vulnerability Notes Database's top 20 highest scoring vulnerabilities. The vulnerability which is not protected (CVE-2001-0333) is because it is based on a logic error that is reproduced in the variation methods we studied. Figure 2 provides a comparison for an MVEE composed of variants using instruction set randomization and library entry point randomization.

	Total Vulnerabilities	Arbitrary Code Execution Vulnerabilities	Number Protected
Apache	33	3	3 (100%)
MySQL	29	3	3 (100%)
BIND	10	1	1 (100%)
CERT Top 20	20	15	14 (93%)

Figure 2: Amount of coverage provided by an optimal two-variant MVEE (instruction set randomization and library entry point randomization) against known vulnerabilities from various sources.

When using a two-variant MVEE in a honeypot context, an effective combination is the use of many forms of address space randomization (heap layout, stack base, library entry point) as possible in one variant and either system call number randomization or instruction set randomization in another. The differing layers of address randomization are effective at detecting the different kinds of memory related attack vectors, and system call number randomization or instruction set randomization variant detects unknown attacks.

In a three-variant MVEE, an unmodified executable is quite useful. An unmodified executable in this situation becomes a "canary" variant, because it is the closest match to a binary found on a regular system. The post-mortem analysis mentioned in the previous section is performed on this variant because the monitor will not have to reverse the effects of one or more variation techniques.

While performance is not a significant issue with running honeypots, extremely slow honeypots are not enticing to attackers. This means that any software running on the monitor at either level of monitoring granularity should not cause network timeouts or other significant delays due to the monitor's processing. Instruction set randomization, for example, incurs a large performance penalty due to the decoding of every instruction. This makes instruction set randomization infeasible as a variation technique for this purpose unless modern hardware is used. Detection of shellcode in buffers of various length is a widely studied topic. Network-based intrusion detection systems are built around a similar kind of feature to detect attacks in network traffic on high bandwidth connections. Therefore, while there is a delay involved with detection, it will not be significant enough to render a honeypot ineffective.

Any combination of variant techniques does not provide any *a priori* indication on the amount of effort required to create an exploit that will be successful against a known set of variations. When multiple variants are used at the same

time, constructing an attack that is successful becomes exponentially more difficult as more variants are added. For example, if an attacker is targeting a three-variant MVEE, a successful attack has to be designed to exploit vulnerabilities in all three variations at the same time, or be able to adequately mimic proper execution of the target program for a time long enough that the other variants can be exploited, while being sufficiently robust to not be damaged by the exploitation of the other variants. Any attack of the latter nature requires passing through at least one synchronization point and intimate knowledge of how the target appears to the monitor, which also raises the complexity of such an attack. Consequently, the likelihood that any dynamic instruction sequence in an exploit that was executed in a reasonable time interval could be created is extraordinarily small. We expect that the time and resources required to construct an exploit with the required complexity would be extremely prohibitive and sufficient to convince an attacker to focus their efforts on easier targets. Honeypots, however, are designed to resemble easy targets to an attacker. An MVEE honeypot presents itself to the outside world as a regular computer, and does not divulge the types of variations used. This prevents the attacker from knowing how to construct an attack against an MVEE without inside knowledge.

6. RELATED WORK

The idea of using diversity to improve robustness has a long history in the fault tolerance community [3]. The basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a program, developed by independent teams), with the hope that they fail independently.

Cox et al. use address space randomization and instruction set randomization to create different variants [6]. With address space randomization, heap and stack of their variant have the same structure as the original program, but they are located at disjunct addresses, i.e., a valid pointer of the original program is never valid in the variant. Instruction set randomization adds a prefix code before each instruction. As an extension of this system, Nguyen-Tuong et al. generalize the idea to any kind of data diversity [9], where a reversible but otherwise arbitrary function is used to encode certain data values.

Bruschi et al. also diversify the memory layout to defeat memory error exploits [5]. They use the same idea as Cox et al. for address space randomization, and extend it with a defense for overwriting only the lower bits of an address. Berger et al. proposes redundant execution with multiple variants to provide probabilistic memory safety by randomizing the object layout across the heap [4]. By using a modified memory allocator, objects are located at different addresses in the variants.

Our prototype implementation of multi-variant execution [10] uses a userspace monitor that synchronizes variants at the system call level. The monitor is able to mitigate causes of false positives such as signals and random numbers, and is well suited for multi-core machines. Other variants include an implementation of system call number randomization, and the combination of reverse stack execution and system call number randomization in the same executable.

7. CONCLUSION

A multi-variant execution environment significantly reduces the probability of successful attacks and execution of arbitrary code. An optimal two-variant MVEE has enough coverage to stop arbitrary code execution attempts, and a three-variant MVEE allows for a thorough reconstruction of a binary commonly found on production machines. Based on these properties, a honeypot does not need to emulate vulnerabilities while leaving the target software running with full capabilities. This limits exposure to attacks and facilitates a wider data collection ability because emulation layers do not have to be written. The low processing requirement permits deployment in virtual machines or older hardware. The flexible nature of the MVEE gives security professionals the ability to make a more informed decision and more carefully weigh the costs and benefits of using different variation techniques.

For future work, we are interested in creating objectively measurable standards for the effectiveness of multi-variant execution environments and variation techniques. By analyzing historical vulnerabilities and legacy code, we strive to extrapolate the ability of MVEEs to withstand the attacks of the future and design new variation techniques.

8. REFERENCES

[1] Dionaea. http://dionaea.carnivore.it.

[2] Nepenthes. http://nepenthes.carnivore.it.

[3] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the International Computer Software and Applications Conference*, pages 149–155. IEEE Computer Society, 1977.

[4] E. Berger and B. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.

[5] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicae for defeating memory error exploits. In *Proceedings of the International Workshop on Information Assurance*, pages 434–441. IEEE Computer Society, 2007.

[6] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the USENIX Security Symposium*, pages 105–120. USENIX Association, 2006.

[7] National Institute of Standards and Technologies. *National Vulnerability Database*, 2009. http://nvd.nist.gov.

[8] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[9] A. Nguyen-Tuong, D. Evans, J. Knight, B. Cox, and J. Davidson. Security through redundant data diversity. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 187–196. IEEE Computer Society, 2008.

[10] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the European Conference on Computer Systems*, 2009.

[11] K. Scott, , K. Scott, and J. Davidson. Strata: A software dynamic translation infrastructure. In *In IEEE Workshop on Binary Translation*, 2001.