CS 5352: Advanced operating Systems Design

Report on Paper: Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization

Ravi Sankar Gogineni
R11788968

Overview of the Paper:

In recent years, the deployment of many applicationlevel countermeasures against memory errors and the increasing number of vulnerabilities discovered in the kernel has fostered a renewed interest in kernel-level exploitation. Unfortunately, no comprehensive and well established mechanism exists to protect the operating system from arbitrary attacks, due to the relatively new development of the area and the challenges involved. In this paper, we propose the first design for finegrained address space randomization (ASR) inside the operating system (OS), providing an efficient and comprehensive countermeasure against classic and emerging attacks, such as return-oriented programming. To motivate our design, we investigate the differences with application-level ASR and find that some of the well-established assumptions in existing solutions are no longer valid inside the OS; above all, perhaps, that information leakage becomes a major concern in the new context. We show that our ASR strategy outperforms stateof-the-art solutions in terms of both performance and security without affecting the software distribution model. Finally, we present the first comprehensive live re-randomization strategy, which we found to be particularly important inside the OS. Experimental results demonstrate that our techniques yield low run-time performance overhead (less than 5% on average on both SPEC and sys call-intensive benchmarks) and limited run-time memory footprint increase (around 15% during the execution of our benchmarks). We believe our techniques can greatly enhance the level of OS security without compromising the performance and reliability of the OS.

The contributions of the paper are as follows:

The contributions of this paper are threefold. First, we identify the challenges and the key requirements for a comprehensive OS-level ASR solution. We show that a number of assumptions in existing solutions are no longer valid inside the OS, due to the more constrained environment and the different attack models. Second, we present the first design for finegrained ASR for operating systems. Our approach addresses all the challenges considered and improves existing ASR solutions in terms of both performance and security, especially in light of emerging ROP-based attacks. In addition, we consider the application of our design to component-based OS architectures, presenting a fully-fledged prototype system and discussing real-world applications of our ASR technique. Finally, we present the first generic live re-randomization strategy, particularly central in our design. Unlike existing techniques, our strategy is based on run-time state migration and can transparently re-randomize arbitrary code and data with no state loss. In addition, our re-randomization code runs completely sandboxed. Any run-time error at re-randomization time simply results in restoring normal execution without endangering the reliability of the OS.

Related Work:

Prior work on ASR focuses on randomizing the memory layout of user programs, with solutions based on kernel support, linker support, compiler-based techniques, and binary rewriting. A number of studies have investigated attacks against poorly-randomized programs, including brute forcing, partial existing approaches can support state-full live re-randomization. The general idea of randomization has also been applied to instruction sets, data representation, data structures, memory allocators. Our struct layout randomization is similar to the one presented, but our ASR design generalizes this strategy to the internal layout of any memory object (including code) and also allows live layout re-randomization. Finally, randomization as a general form of diversification has been proposed to execute multiple program variants in parallel and detect attacks from divergent behaviour. Unlike our solution, none of the existing ASR techniques can support live re-randomization with no state loss. For live re-randomization Prior work that comes closest to our live re-randomization technique is in the general area of dynamic software updating. Many solutions have been proposed to apply run-time updates to user programs and operating systems. Our re-randomization technique shares with these solutions the ability to modify code and data of a running system without service interruption. The fundamental difference is that these solutions apply run-time changes in place, essentially assuming a fixed memory layout where any state transformation is completely delegated to the programmer. Our solution, in contrast, is generic and automated, and can seamlessly support arbitrary memory layout transformations between variants at runtime. Other solutions have proposed process-level run-time updates to release some of the assumptions on the memory layout, but they still delegate the state transfer process completely to the programmer. This completely hinders their applicability in live re-randomization scenarios where arbitrary layout transformations are allowed.

Why is better than other techniques?

- the deployment of many application level countermeasures against memory errors and the increasing number of vulnerabilities discovered in the kernel has fostered a renewed interest in kernel-level exploitation.
- the first design for fine-grained address space randomization (ASR) inside the operating system (OS), providing an efficient and comprehensive countermeasure against classic and emerging attacks, such as return-oriented programming
- Kernel-level exploitation increasingly gaining momentum. Many exploits available for Windows, Linux, BSD, Mac OS X, iOS.
- Plenty of memory error vulnerabilities to choose from. Plethora of internet-connected users running the same kernel version. Many attack opportunities for both local and remote exploits.

Address Space Randomization:

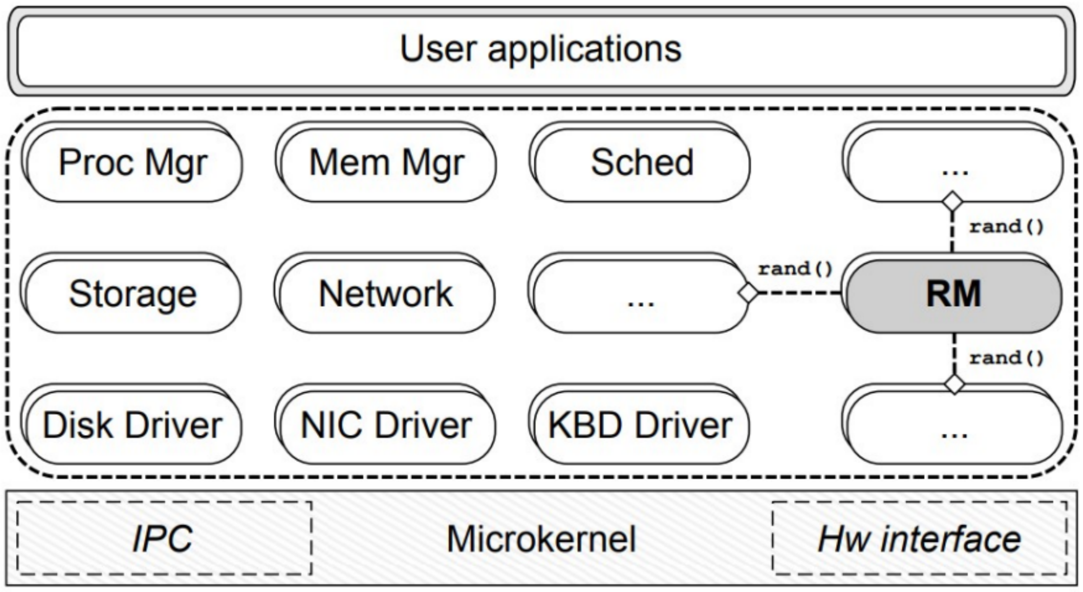
- Well-established defense mechanism against memory error exploits.
- Application-level support in all the major operating systems.
- system itself The operating typically not randomized at all.
- Only recent Windows releases perform basic text randomization.
- Goal: Fine-grained ASR for operating systems.

A Design in OS-Level ASR:

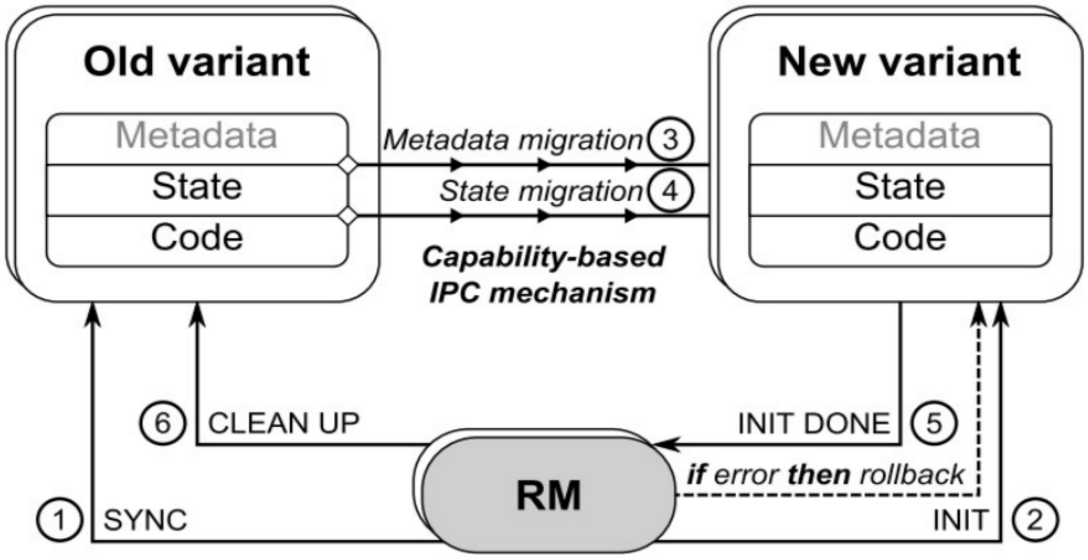
- Make both location and layout of memory objects unpredictable.

- Minimal amount of untrusted code exposed to the runtime.
- Live re-randomization to maximize un-observability of the system.
- No changes in the software distribution model.

Architecture:



ASRR Transformations:



We have implemented our ASR design on the MINIX 3 microkernel-based operating system, which already guarantees process-based isolation for all the core operating system components. The OS is x86-based and exposes a complete POSIX interface to user applications. We have heavily modified and redesigned the original OS to implement support for our ASR techniques for all the possible OS processes. The resulting operating system comprises a total of 20 OS processes, including process management, memory management, storage and network stack. Subsequently, we have applied our ASR transformations to the system and evaluated the resulting solution.

Performance:

To evaluate the performance of our ASR technique, we ported the C programs in the SPEC CPU 2006 bench mark suite to our prototype system. We also put together a dev tools macro benchmark, which emulates a typical system call-intensive workload with the following operations performed on the OS source tree: compilation, find, grep, copying, and deleting. We performed repeated experiments on a workstation equipped with a 12-core 1.9Ghz AMD Opteron "Magny-Cours" processor and 4GB of RAM, and averaged the results. All the OS code and our benchmarks were compiled using Clang/LLVM 2.8 with -O2 optimization level. To thoroughly stress the system and identify all the possible bottlenecks, we instrumented both the OS and the benchmarks using the same transformation in each run. The default padding strategy used in the experiments extends the memory occupancy of every state object or struct member by 0- 30%, similar to the default values suggested.

Effectiveness:

- As pointed out in, an analytical analysis is more general and effective than empirical evaluation in measuring the effectiveness of ASR.
- The effectiveness of ASR techniques against guessing and brute-force attacks. These attacks are far less attractive inside the operating system.
- The internal layout randomization provides better protection, forcing the attacker to learn the relative distance/alignment between two memory elements in the general case.
- This technique constraints attacks based on arbitrary memory reads/writes to learn the address of the target element.
- ASR design can also be used as the first "live-workaround" system for security vulnerabilities.

Conclusion:

- The first ASR design for operating systems to fully explore the design space, we presented an analysis of the different constraints and attack models inside the OS, while highlighting the challenges of OS-level ASR.
- The technique can also be applied to generic user programs, improving existing application-level techniques in terms of both performance and security, and opening up opportunities for third-generation ASR systems.
- This strategy is more portable and much safer than existing techniques, which either rely on complex binary rewriting or require a substantial amount of untrusted code exposed to the runtime.
- In this technique, the complex re-randomization code runs completely sandboxed and any