1 Summary

Exploits against software vulnerabilities is the most popular attack vector to compromise computer systems. While much effort has been spent on designing, building, and deploying software that is free of defects, software systems of even modest complexity are still routinely deployed with vulnerabilities. More alarmingly, even the trusted computing base (e.g. OS kernel) may contain vulnerabilities that would allow attackers to subvert security mechanisms like the application sandbox on smartphones.

My research aims to provide principled and practical solutions to prevent exploits against software vulnerabilities. I believe a security solution must be principled, otherwise it will not be able to withstand the rapidly evolving arm race from offensive technologies. At the same time, I also believe that, as a discipline of applied science, the most important criterion to evaluate the quality of security research is its practical impact; therefore, a security solution must also be practical.

In achieving my research goal, I mainly focus on three complementary areas: (i) automated patching and hardening, for vulnerabilities that can be fixed in generic ways, I have developed automated systems that can diagnosis and patch integer overflow vulnerabilities [12], eliminate use-after-free vulnerabilities [4], and prevent bad-casting vulnerabilities [5]; (ii) runtime exploit mitigation, for vulnerabilities that cannot be eliminated with practical runtime overhead (e.g. memory safety bugs), I have built efficient exploit prevention techniques that can fundamentally mitigating code injection attacks [10], code reuse attacks [6,13,14], and non-control-data attacks [11]; and (iii) vulnerability discovery, for vulnerabilities that still require manual patching, such as semantic and logical errors, I have designed analysis techniques to discover vulnerabilities in system designs [2,3], alternative I/O paths [1], and file systems [7]. When tackling these challenges, I am always fascinated by the power of interdisciplinary research. And many of the above projects have benefited from collaborations with researchers from other areas computer science, including operating system, program analysis, and architecture.

My research has received the 2015 Internet Defense Prize by Facebook and USENIX, and 2015 NYU-Poly Best Applied Security Research Paper Award. Many of them were also covered by high-profile media such as MIT Technology Review, Ars Technica, ZDNet, Dark Reading by InformationWeek, etc.

2 Dissertation Research

Due to the popularity of C/C++ programs, memory corruption bugs are prevalent. Once exploited, memory corruption bugs also grant attackers great power over the software systems, such as reading and writing arbitrary memory address. For these reasons, memory corruption based exploits have been one of the most popular attack vectors since the 1988 Morris worm. My thesis work thus focuses on preventing memory corruption based exploits.

Preventing Code Injection Attacks via JIT Engines. The very first exploit technique utilized by attackers is code injection attack. Although it is widely believed that code injection attacks have already been eliminated by mechanisms like DEP (data execution prevention), my study showed that this exploit technique can be revived through abusing dynamic code generators (DCG), such as JavaScript engines in modern browsers [10]. Specifically, to speed up the execution, DCG keeps generated native code in a code cache, which is both executable and writable (for the purpose of code emitting, patching, and garbage collection). This makes the code cache a perfect target for injecting malicious shellcode. Furthermore, I demonstrated that such attacks are non-trivial to prevent. In particular, some DCGs have tried to protect the code cache by switching its access permission between RX (executable but not writable) and WR (writable but not executable), but in a multi-threading execution environment, such protection can be bypassed by leveraging two concurrent threads.

To fundamentally prevent such attacks, I designed and built secure dynamic code generation (SDCG), a new architecture that (i) enables DCGs to comply with the W⊕X policy; (ii) eliminates the race condition in multi-thread execution environment; (iii) is easily to adopt; and (iv) introduces small performance overhead [10]. SDCG achieves these goals through a multi-process-based architecture. Specifically, instead of generating and modifying code in a single process, SDCG relocates the DCG functionality to a second trusted process. The code cache is built upon memory shared between the original process and the trusted process. In the original process, the code cache is mapped as RX; and in the trusted process, the same memory is mapped as WR. By doing so, the code cache remains read-only under all circumstances in the untrusted process. Thus the race condition that allows the code cache to be writable to untrusted thread(s) is eliminated. At the same time, the code generator in the trusted process can perform code generation, patching, and garbage collection as usual. To enable transparent interaction between the code generator and the generated code, SDCG only requires adding a few wrappers that make the code generator invocable through remote procedure calls (RPC). As a demonstration, I have ported two typical code generators to this new architecture: one dynamic binary translator (Strata) and one JavaScript engine (Google V8). Evaluation showed that the performance overhead imposed by SDCG is small: around 2% for Strata, and around 6% for V8.
Preventing Privilege Escalation Attacks Against OS Kernel. While most existing memory corruption based attacks aim to hijack the control flow of a program, non-control-data attacks are also feasible and powerful, especially in OS kernels. As a demonstration, I have developed an exploit that only attacks non-control-data yet is able to achieve full unthethered jailbreak (i.e. privilege escalation) on Android devices. Not only did I demonstrate the attack, but I also designed Kenali [11], a principled and practical approach to defeat such attacks. Specifically, memory corruption based privilege escalation attacks are essentially violations of two security principles for access control mechanisms (a.k.a. reference monitors): (i) complete mediation: attackers should not be able to bypass designated access control checks; and (ii) tamper proof: attackers should not be able to tamper with the integrity of either the code or data of the reference monitor. Based on this observation, Kenali eliminates privilege escalation attacks via enforcing these two security invariants.

To achieve this goal, Kenali utilizes the data-flow integrity (DFI) technique. Similar to control-flow integrity (CFI), DFI guarantees that runtime data-flow does not deviate from the data-flow graph generated from static analysis. For example, the data from a string buffer should never flow to the return address on stack (control-data), or to the uid (non-control-data). However, enforcing DFI for the entire kernel is not practical for its high performance overhead. To overcome this challenge, I designed two new techniques. The first technique leverages static program analysis to discover all critical data that can affect kernel access control checks. The key observation is that, although access control checks are scattered throughout the kernel, they all follow one semantic rule, i.e. if a security check fails, it must return a security related error code. Combining this observation with standard program analysis techniques like control- and data-dependency analysis, Kenali can soundly discover all access control data without manual annotation. Taking the analysis result as input, the second technique enforces selective DFI over the security critical data with a two-layer protection scheme: the first layer is a light-weight data-flow isolation mechanism that prevents irrelevant data-flow from tampering critical data; then the second layer enforces the more expensive DFI to prevent illegal data-flow within critical data.

To maximize the practical impact, I have implemented a prototype of Kenali for Android devices, because (i) Android is the most popular open sourced platform; and (ii) due to its long updating cycle, exploit prevention techniques play a more important role for Android devices. Evaluation over the prototype implementation showed that Kenali was able to stop various of privilege escalation attacks, and imposed moderate performance overhead.

Improving Efficiency with New Hardware Support. During the development of above defense systems, I was constantly troubled by the fact that software-based approaches are not efficient enough and hardware-based approaches are not flexible enough. To solve this problem, I designed hardware-assistant data-flow isolation \(\text{HDFI}\), a new fine-grained data isolation mechanism that is broadly applicable and very efficient. HDFI enforces isolation at the machine word granularity by virtually extending each memory unit with an additional tag that is defined by data-flow. This capability allows HDFI to enforce a variety of security models such as the Biba Model for data integrity and the Bell–LaPadula Model for data confidentiality. For demonstrating its benefit, I developed and ported several security mechanisms to leverage HDFI, including stack protection, standard library enhancement, virtual function table protection, code pointer protection, kernel data protection, and information leak prevention. My experience showed that HDFI is easy to use and usually allowed me to create more elegant and more secure solutions. We have implemented HDFI by extending the RISC-V instruction set architecture (ISA) and instantiated it on the Xilinx FPGA evaluation board. Evaluation results showed that the performance overhead caused by our modification to the hardware is low (\(< 2\%\)) and the security applications I implemented impose low performance overhead. This work is currently under submission.

3 Other Projects

Preventing Code Reuse Attacks. Since code reuse attacks are becoming more and more popular, I have also collaborated with other researchers on stopping such attacks. The de facto security mechanism to stop code reuse attacks is address space layout randomization (ASLR), which aims to prevent attackers from locating required code gadgets. In theory, ASLR can complement W⊕X and stop code reuse attacks effectively. However, due to information leak vulnerabilities (e.g. memory corruption based arbitrary read), ASLR is usually bypassable. To solve this problem, we first conducted a systematic study to identify all possible ways to derive the location of a code gadget. Based on the result of this study, we designed and built ASLR-Guard, a system that completely prevents code address leak to stop code reuse attacks [6]. In particular, ASLR-Guard utilizes three complementary techniques: (i) it completely decouples code and data by remapping all code to random addresses, so data pointers can no longer be used to infer the location of code; (ii) it stores all code pointers in a secure memory region; and (iii) whenever a code pointer is going to be propagated to regular memory regions outside the secure region, ASLR-Guard leverages a novel encryption scheme to encrypt the pointer. My main contribution to this project, the encryption scheme, is secure as it never leaves plaintext code pointers in regular memory or registers and includes integrity checks to defeat forgery attacks. It is also very efficient, so ASLR-Guard incurs almost no runtime overhead (\(< 1\%\)).

Besides compromising code pointers, another emerging attack target for code reuse attacks is virtual function table (VTable) pointers. To defeat such attacks, we have developed two systems: VTint [13] aims to prevent VTable hijacking attacks for commodity binaries; and VTrust [14] provides more thorough protection by leveraging source code based instrumentation.
Automated Program Hardening. Patching vulnerabilities is not trivial, so developers often make incorrect patches. To help reduce such errors, I have developed several systems that can automatically harden programs to remove bugs. For example, due to the gap between high-level program languages and machine instructions, integer overflow vulnerabilities are notorious for patching. To solve this problem, we have implemented SoupInt [12], a system that can automatically diagnosis integer overflow vulnerabilities and generate emergency patches. Another vulnerability class that is hard to diagnosis and patch is use-after-free, as it requires deep understanding of why the memory is mistakenly freed. To solve this problem, we have developed DangNull, an automated hardening system to eliminate use-after-free vulnerabilities. Similarly, we have developed Caver, an automated hardening system to prevent bad-casting bugs, another common vulnerability class for C++ programs. This work was awarded the 2015 Internet Defense Prize from Facebook and USENIX.

Vulnerability Discovery. For vulnerability discovery, my research focuses on high-level bugs. For example, by performing cross-comparison among different implementations of the same functionality, our system Juxta [7] discovered over one hundred semantic errors in Linux file systems. Using a similar methodology, we found that while many mechanisms like UAC (User Access Control) have been introduced to authenticate that the input is truly from the user, lacking corresponding checks in alternative I/O paths (e.g. accessibility interface) allows attackers to bypass those security mechanisms [1]. We also discovered design flaws that allow attackers to bypass ASLR [3], and stealthily inject malware into iOS devices through a malicious charger [2]. The practical significance of these discoveries are recognized by top academic conference like SOSP and CCS, as well as top industrial conference BlackHat.

Malware Related. My security research career started with malware related problems. During that time, I have built (i) honeypot-based system to collect malware [15]; (ii) automated analysis system to extract information about criminal infrastructures and underground economy activities from the collected malware samples [16]; and (iii) runtime monitoring system to detect and prevent drive-by download attacks [8]. While I believe this research direction is still very valuable for understanding attackers, a later study I conducted showed that the collecting – analyzing – signature extraction – detection pattern, though is still widely used by anti-virus industry, does not provide distinguishing advantage over malware authors [9]. For this reason, my more recent research has shifted to eliminating the main spread vector of malware, i.e. exploit against software vulnerabilities.

4 Future Directions

My long-term research goal is to create trustworthy computing systems. In pursuit of this goal, I will continue working on challenging problems in the area of systems and software security. Below are several problems that are related to this direction.

Eliminating Memory Corruption Bugs. More often than not, security and privacy take a back seat to the rapid deployment of new and interesting functionalities. Take memory safety as an example, because existing techniques either require developers to manually retrofit legacy code (e.g. Ccured) or impose high performance overhead (e.g. SoftBound), none of them are adopted in practice. However, I believe that with better optimization techniques and new hardware features (e.g. Intel MPX), it is possible to completely eliminate (exploitable if not all) memory corruption bugs in commodity C/C++ software.

Detecting Errors in Security Mechanisms. Although security mechanisms like access control are assumed as part of the trusted computing based, in practice, I have discovered many vulnerabilities inside them. Some of these vulnerabilities are caused by design flaws, while others are introduced by implementation errors. Similar to formal verification of security policies, I want to address these problems with software verification technologies. The ultimate goal is to provide end-to-end safety guarantees, i.e. the design satisfies expected safety properties and the implementation meets the specification.

Better Access Control Mechanisms. With above technologies laying a solid foundation, the next step is to build better access control mechanisms. My observation is that, from enterprise to individual Internet users, the real value lies in data, so it is important to protect their data from unauthorized access. The challenge is that asking average users to understand the security and privacy consequences of every action they take is impractical. At the same time, as we are making exploits against software vulnerabilities more and more difficult, attackers are likely to move to easier targets, such as exploiting unwary users through social engineering. For these reasons, we need better access control mechanisms. I envision two promising directions. First, I want to develop more intuitive access control mechanisms. A good example along this direction is the user-driven access control, where authorization to security and privacy sensitive resources is embedded transparently as part of user interaction, e.g. only when the user click the camera button does the system authorize an app to access the camera. By doing so, we can provide users more controls over their data while keep the security mechanism transparent. However, there are still many open problems in this area that current approaches have difficulties to handle.

Secondly, many social engineering attacks can be abstracted as mismatch between what user sees (and expects) and what the program actually does. Therefore, I am interested in utilizing machine learning technologies, such as computer vision and natural language processing to model what a user expects and only grants permissions to resources that are implied by such expectations.
References


[compiled on 2016-01-09]