Research Statement

Brendan Dolan-Gavitt

Research in computer security is more important than ever. Software has expanded into nearly every aspect of modern life, from cars to coffee makers. This expansion into new areas has not been accompanied by safer programming practices, and the safety and reliability of these devices is highly suspect. At the same time, powerful adversaries such as national governments and organized crime have displayed both motivation and aptitude for exploiting weak systems. These circumstances make it imperative that we find effective ways of securing existing systems and develop new architectures for building secure systems.

My own research focuses on *understanding the behavior of systems and software for security*. This includes the use of automated techniques to understand what real software is doing, and how to discover implicit and undocumented assumptions in systems. In short, I want to minimize the human effort required to understand the architecture, organization, and internals of computing systems, and use that understanding to produce effective ways of defending them against attackers.

Thesis Work

My research over the past few years has led to several new ways of automatically understanding the behavior of large, real-world systems and producing novel defenses against attacks. In this section, I will describe my key publications and their impact on the state of the art.

In **Robust Signatures for Kernel Data Structures** [3], I developed techniques for probing the invariants enforced on kernel-mode data structures. These structures represent objects of significant interest from a security perspective, as they include files, processes, threads, and network connections. Because of this, tools often locate kernel objects by scanning memory

using invariants on data structure fields. However, this strategy only works if the invariants are *enforced*—if an attacker can manipulate the data in an object, they can hide it from scans and evade detection. Indeed, I found that the invariants that were checked by memory scanners were rarely enforced by the kernel, presenting many opportunities for attackers to hide. To correct this situation I created a novel dynamic analysis that combines *profiling* (checking how often each field in a structure was accessed by the operating system) with *fuzzing* (actively making changes to the structure fields and observing the OS's response) to find out which fields are strictly checked by the OS. Finally, we generated new, *robust* signatures by finding invariants on precisely those fields that were most strictly checked by the kernel itself, and that an attacker would have the most difficulty in tampering with. Our robust signatures effectively resist evasion even by powerful adversaries that have the ability to manipulate dynamic kernel data and make it impossible for attackers to hide objects from memory scans.

I have also made significant contributions to the field of *virtualization security*. In this area, the goal is to harden systems against attack by separating security software from the systems they are meant to protect by placing them into separate virtual machines (typically referred to as the *security* and *guest* virtual machines, respectively), with a small and verifiable hypervisor ensuring the isolation between the two. Thus, even if the operating system running inside the VM is compromised, hypervisor-level monitoring will remain secure and will be able to respond to the attack effectively. A key challenge in this area is the *semantic gap*: although the separation between the two virtual machines means that tools in the security VM are resistant to tampering, visibility into the guest VM is greatly reduced. Virtualization security tools are presented with a low-level view of the guest VM that is typically limited to the contents of RAM and the CPU registers, but to make effective security decisions they need information about high-level OS concepts such as processes, threads, files, and network connections. Thus, virtualization security tools must painstakingly reconstruct a high-level view from this low-level data, which requires deep understanding of the algorithms and data structures of the OS and applications running in the guest VM. Worse, in the case of closed-source operating systems this understanding can only be derived by time-consuming and expensive manual reverse engineering of the binary code. The semantic gap therefore poses a significant obstacle to the development and deployment of virtualization security.

In **Virtuoso: Narrowing the Semantic Gap in Virtual Machine**

Introspection [2], I made the first significant progress on the semantic gap problem in virtualization security. The critical insight is that the operating system must already contain code that does the work of "parsing" out low-level data into higher-level abstractions—after all, this is precisely how public-facing APIs and system administration tools work. To capitalize on this idea, I developed novel techniques (based on *dynamic program slicing*) that analyze whole-system execution traces of programs performing tasks such as listing processes, extract out just the code that computes the list of processes, and then transforms it into a program that perform the same task with access only to a physical memory image and CPU state. This work made virtualization-based security monitors practical by transforming a manual process that required weeks of expert manual effort into one that needs just a few minutes of computation time.

A related problem arises when considering virtualized security systems that need to do more than passive monitoring of a system. In cases where an intrusion detection system such as antivirus software must receive notification when certain events happen such as a file being opened or a URL being visited in a web browser, careful analysis of the guest operating system is needed to determine where to interpose. As with the semantic gap problem, this once required many painstaking hours with a disassembler to locate the functions (often private and undocumented) that corresponded to the event of interest. In **Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection** [1], I found that by grouping the memory accesses made by a system according to the code that generated them, it was possible to apply techniques from information retrieval and machine learning to locate code that corresponded to events of security interest. This allowed us to quickly find points at which to interpose on a system for security monitoring on five operating systems and two processor architectures with no reverse engineering required. This work was also the first to consider program analysis as a "big data" question, where executing code generates reams of data that can then be aggregated and searched for patterns of interest.

Future Work

Looking broadly at the goals of security and privacy in computing systems, I see two areas where significant progress must still be made. First, many current systems are difficult to analyze and model by security researchers

because they have been built with the assumption that their internals are no one's business but the original developers. As my past work demonstrates, this assumption is dangerously wrong. In addition to developing ever-more powerful ways to tease out the hidden implementation details of hardware and software, we might instead turn the question around and ask whether we can construct programming languages and tools that expose the internals of a program in a self-documenting way. For example, a compiler might be able to automatically produce parsers for the in-memory data structures used by a program, or generate a list of useful hook points throughout the code, such as places where data crosses marked security boundaries.

We might even extend this idea to hardware design, and ask whether there are ways to make currently obscure and undocumented hardware self-documenting. For example, one could imagine tools that, in addition to helping lay out an embedded peripheral, also create code for an emulator model of that peripheral. This would allow one to have virtual instances of a full embedded device "for free," allowing much larger scale testing of device code (it is, after all, much easier to run a thousand virtual machines than it is to obtain and test a thousand internet-enabled coffee makers). Moreover, making such models available to third party developers would make it possible for other operating systems to be ported to embedded platforms, giving users much more freedom and control over the gadgets that increasingly dominate their lives.

Second, commodity computing systems, be they desktop operating systems, mobile phones, or low-powered embedded platforms, are generally entirely opaque to the user. As such, they contain innumerable opportunities for mischief to go unnoticed; this manifests itself both in the prevalence of backdoors in commercial software and the ease with which malware runs undetected on end users' systems. In addition to making implementation details available to developers, I plan to investigate how we might make system execution understandable to end users. This would require deep changes to the way we construct systems, as we would need a way to tie its low-level implementation to a high-level semantic description of its runtime behavior in a way that is *verifiable* (it would do no good if programs could claim malicious behavior was something innocent). We can summarize this research path by asking whether we can ensure that a program 1) does what it says; 2) says what it does; and 3) can prove it.

In the absence of cooperation from hardware or software developers, we can also ask whether it is still possible to make progress on these research

goals. For example, could we find ways of automating the creation of hardware models? Can we create high-level summaries of the behavior of software on a system? Although these tasks are likely to be impossible in the general case (due to fundamental limitations such as the Halting Problem), we may be able to find real-world classes of programs or devices for which we can accomplish our goals.

References

[1] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee. Tappan Zee (North) Bridge: Mining memory accesses for introspection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[2] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[3] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.