

My research interests are in the areas of software systems and security. For my Ph.D. research, I've been looking at methods to fortify software applications and the operating system to withstand increasingly sophisticated attacks by black-hat hackers. To accomplish these tasks, I have modified various open-source systems like the Linux kernel, the GNU C compiler and the `bochs x86` emulator. In doing so, I have also acquired deep knowledge of the software run-time environment on Linux, including its process memory model and mechanisms for dynamic linking and system-call handling, and of program compilation. Learning about the internal workings of software systems, and discovering ways of evolving them to create immunity against different attack techniques has been a rewarding experience. Below, I briefly describe my major research accomplishments categorised by area.

Research Summary

Fortifying the Operating System

In this research, I recognised the powerful vantage point of the operating system as the manager of all system resources on a machine. Processes access virtually all system resources, e.g., hard disk usage, network communications, inter-process messaging, by making system-call invocations into the operating system kernel. This implies that any attack code that is executing on a compromised process will, too, need to invoke system call(s) to cause any out-of-process damage.

I found the system-call interface between user-mode programs and the kernel running in supervisor mode to be a great location to establish a monitoring mechanism for these invocations. As part of my work on e-NeXSh[8], I implemented a lightweight system-call interception mechanism in the system-call handler of `linux-2.4.18-3`. This can detect the execution of malicious code that has been injected into the program memory as soon as it invokes a system call. To counter return-into-libc attacks, I implemented a wrapper mechanism for functions in the Standard C library. I use this wrapper mechanism to reconstruct the current program call stack, and validate all activation records on the program stack against the binary image of the executable. My combined monitoring of LIBC function- and system-call invocations made by processes can efficiently detect any deviations in the observed behaviour of running programs from what is normal — normal, i.e., *anticipated* behaviour constitutes of invocations of LIBC functions and system calls only from code addresses that exactly match corresponding call sites in the program image. This makes e-NeXSh a very simple, low-overhead OS-fortifying technique that utilises information about the process run-time to accurately determine when a process has been compromised.

In RiSA[9], using the `bochs x86` emulator, I developed a “Randomised Instruction Set Architecture” machine designed to protect software against any code-injection attack by creating unique, per-process machine instruction sets. Each randomised instruction set had its own unique bit-pattern to denote the same `x86` machine instruction, meaning that a byte-sequence representing unmodified `x86` machine instructions would execute in a uniquely different manner for each randomised instruction-set. Furthermore, such a byte-sequence would contain an opcode that is invalid for the randomised set, and be terminated by the operating system, causing the attack attempt to be thwarted.

Given that e-NeXSh and RiSA are implemented within the OS kernel (or hardware), these techniques have the advantage that they don't require applications to be modified, and can be applied to software for which no source code is available. While both techniques successfully defeat

code-injection attacks (e-NeXSh handles return-into-libc attacks as well), e-NeXSh has greater practical significance due to its really low execution overhead.

Programming Languages, Compilers

Prior to my OS-based security research, I worked on the concept of “Security-Optimising Compilation” — enhancing the task of program compilation to build in automatic security checks. In Casper[7], I modified the GCC compiler to produce instrumented code that resists attack techniques involving the overwriting of the return address stored in functions’ activation records on the stack, e.g., Aleph One’s classic Stack-smashing technique[3]. The compiler-introduced instrumentation protects the stored return address in activation records by detecting any modifications made as a result of stack-based buffer overflows or any format-string exploits.

During the summer of 2003, I interned with the Internet Security Group at IBM Research, Hawthorne, that was developing with a high-assurance, secure operating system for a smart-card environment. My assigned responsibility was to re-target the GCC compiler, making it generate assembly (from C code) for the SmartXA smart-card processor. We had to extend the scope of this project to handle C language extensions that were used in the operating system code. This experience was invaluable in how it helped me better understand both the process of compilation as it maps the same source code to differing machine architectures and instruction sets, and the inner workings of a commercial-grade compiler.

As an undergraduate at Imperial College, London, I worked on two distinct research projects related to the operational semantics for Java as defined and used by the SLURP group[2]. For the first project, I worked with Dr. Krysia Broda to implement an interpreter (in Prolog) for a subset of the Java semantics definitions, to be used as a theorem prover for proving safety properties about Java. Then, for my undergraduate research project[6], I extended the semantics to also cover both `constructors` and `access modifiers` for Java classes and their member `fields`, `methods` and `constructors`. Learning the details of Java’s language specifications for these projects helped establish my interests in programming languages and their run-time systems.

Distributed Systems: Mobile code for Survivable and Autonomic Systems

From June 2000 to October 2002, I designed and implemented Worklets, a Java-based mobile-code system for use in delivering newly-written code into a running distributed system. The primary objective of Worklets was to seamlessly integrate new code into running systems without requiring a restart. A notable accomplishment in this work was the ability to automatically extract the new byte-code from the sender Java Virtual Machine’s (JVM) memory for transportation into the target JVM. This simplifies the deployment process by eliminating the manual task of having to keep track and explicitly package the various `.class` files (possibly only available at the sender JVM) referenced by the new code. I also supervised M.S. student projects that extended the Worklets system, e.g., integrating cryptographic measures to prevent unauthorised reading/modifications of byte-code during transit, and a “service naming and discovery” mechanism that allows Worklets-enabled hosts to dynamically locate each other on the network.

Worklets was incorporated in two main research areas: the Survivable Workflow System[4] from Naval Research Laboratories (NRL), and Columbia’s Kinesthetics eXtreme (KX) project[5, 10] for Autonomic Systems. The NRL system describes distributed workflow processing utilising multiple software components. However, the failure of any single component would freeze the overall execution of the system. We augmented the system with a Worklets-based control-flow

logic to dynamically acquire and utilise alternate task-processors (components) to resume from wherever components had failed — this enhanced the “survivability” of the NRL system. KX, an ongoing project at the Programming Systems Lab (PSL), Columbia is an add-on, external monitoring architecture for performing run-time analyses and feedback control-based remedial dynamic patching of systems, thus enabling them with autonomic (self-healing, self-modifying) capabilities. The Worklets system was used to both create dynamically insertable probes for collecting data about the running system, and for incorporating run-time patching with new code.

Security Issues with Machine-Readable Travel Documents

During my second internship with the Internet Security Group at IBM Research, Hawthorne, in the summer of 2004, I analysed the cryptographic communications protocols defined for the next generation of electronic, machine-readable travel documents (MRTD)[1] employing biometrics-matching technology, such as passports and visas. My findings revealed that the current specifications for MRTD communications protocols, albeit encrypted, contained security holes and rendered the travel document susceptible to unauthorised data-skimming by an attacker armed with a remote radio frequency (RF) device. I also demonstrated on paper how such attacks can be used to make forged passports that can foil electronic and biometric checks at border posts. I was able to confirm my conclusions with security and cryptography researchers.

This is the first known instance of work on determining the security vulnerabilities of MRTDs — the majority of public interest in the area focuses extensively on the privacy aspects. We expect to present our findings and a security-improvement proposal to the State Department for incorporation into the next iteration of the MRTD specifications.

Future Research

My long-term goals are to explore directions in systems research. I plan to focus on reliability and robustness issues (including security) of software systems. I believe that my background in software and systems in general, including working experience with operating systems, compilers, and programming language run-time environments, puts me in a good position to face this task.

My research agenda immediately after my Ph.D. is to continue research on improving software security against both known and yet-unknown attack techniques. I list below a number of ideas for future exploration — some of these follow naturally from the various works from my dissertation while others are more peripheral in nature, but still very relevant to software security or systems research.

Advanced OS-Hardening Techniques A natural extension of my dissertation work is to evaluate the application of OS-hardening techniques like e-NeXSh to other platforms, e.g., *BSD, Windows XP, AIX, K42, Plan-9. There are interesting research problems in Windows XP due to its microkernel architecture, e.g., system libraries have multiple entry points, making it harder for a grand unified solution to monitor all of them.

Also, e-NeXSh is currently limited in scope and expressiveness to the static image of the program executable — it would make for an interesting problem to extend the LIBC indirection mechanism with more responsibilities. For instance, it can be used to collect training data on feasible parameters for LIBC function invocations, ultimately helping in making e-NeXSh immune against

mimicry attacks that don't cause any changes in control-flow, yet re-use existing program code with malicious parameters.

More! Security-Optimising Compilation techniques for revolutionising compiler design. These changes would produce fortified code with (smarter) automatic run-time checks and also make compile-time program-analysis information available to the run-time for assisting in implementing defence mechanisms. These would lead to more powerful program-execution monitoring techniques for detecting deviations from “normal behaviour,” which in turn are more precisely defined than the simple caller-callee relationships in e-NeXSh.

Network Data Cleanser is a host-based software firewall concept for predicting in real-time, the ill-effects that in-bound data would have on program state, e.g., buffer overflows, or program compromise. I plan to use an abstraction of the data flows for programs being protected to programmatically determine if allowing the data to go through can have any malicious effects. Malicious data that contains executable content can almost certainly be dropped by the firewall, but this can be overridden through the use of sysadmin-defined policies.

Process Rejuvenation is a mechanism with which high-availability requirements of services can be realised. The basic idea is to have timed check-pointing of the process memory and, possibly, external state. Depending on the application's availability requirements, the check-pointing can also be performed at function boundaries, where it may be easier to guarantee the stability of the process memory and external state. The main use for the check-pointing is to allow compromised and/or crashed process to be rejuvenated from the last-known, stable check-pointed state. Other uses include rejuvenation of the process, if compromised by an attacker, in a more secure sandboxed environment (e.g., RiSA). The rejuvenated process can be re-initialised in interpreted or emulated mode to help determine the actual sequence of events leading to the program failure. Another valuable application for this sort of technique is for achieving minimal down-time in mission-critical systems that mandate absolute availability of services.

Mobile Code systems are going to become more valuable in the future given the ubiquity of increasingly powerful mobile devices. A general-purpose mobile code framework that can support automatic downloading of applications on-demand is going to go a long way to fulfill the requirements of the next generation of mobile devices. However, mobile code-related security issues are different from those regarding software security. While software security techniques attempt to prevent foreign code from executing, mobile code security has more stringent requirements in terms of trusting the intentions, or even site of origin, of foreign code. One tentative approach to use specifications for the execution of mobile code, which can then reliably be verified by the host system running the mobile code. I am interested in studying the design of mobile code systems that can scale up to increasing future demands, while still meeting security requirements.

References

- [1] International Civil Aviation Organisation's (ICAO) Machine-Readable Travel Documents (MRTD). <http://www.icao.int/mrtd>.
- [2] SLURP: Sound Languages Underpin Reliable Programming group. <http://slurp.doc.ic.ac.uk/>.
- [3] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
- [4] J.-D. Greze, G. E. Kaiser, and G. S. Kc. Survivor: An Approach for Adding Dependability to Legacy Workflow Systems. Technical Report TR CUCS-026-02, Columbia University, New York, NY, November 2002.
- [5] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto. An Approach to Autonomizing Legacy Systems. In *Proceedings of the Workshop on Self-Healing, Adaptive and Self-MANaged Systems*, June 2002.
- [6] G. S. Kc. Java: Semantics and Type Soundness. Technical report, Imperial College, London, June 1999. B.Eng. Project Report.
- [7] G. S. Kc. CASPER: Compiler-Assisted Securing of Programs at Runtime. Technical Report TR CUCS-025-02, Columbia University, New York, NY, November 2002.
- [8] G. S. Kc. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. Technical report, Columbia University, New York, NY, February 2005.
- [9] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.
- [10] G. Valetto, G. Kaiser, and G. S. Kc. A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems. In *Proceedings of the 8th European Workshop on Software Process Technology*, pages 102–116, June 2001.