Statement of Research Interests

Vinod Ganapathy, January 2007

My research is on *secure and reliable computing* with a strong emphasis on improving software quality. I also maintain an active interest in software engineering, program analysis, formal methods and software verification.

Motivation and Contributions

Conventional wisdom gives us the principle of *Design for Security*: to create secure software, design it to be secure from the ground up. To date, however, only a small fraction of software developed has followed this principle. Diverse security requirements and economic pressures often force software developers to abandon security and focus instead on functionality and performance. As a result, software is deployed with inadequate or non-existent security mechanisms and is thus prone to attacks. Even software originally designed to be secure suffers when code modifications and ever-changing security requirements break key design assumptions [6], thus enabling potential attacks. Our increasing reliance on software infrastructures will only cause the problems posed by insecure legacy software to compound in the future.

My research agenda seeks to mitigate these problems by *securing legacy software.* I have developed techniques both to *analyze* security properties of legacy software and to *transform* it to make it more secure. My research demonstrates that these techniques are effective at improving the security of widely-deployed legacy software. For example, I added authorization checks to the X11 server using the techniques that I developed, and showed that it resisted attacks that were otherwise possible. The following sections describe specific areas in which I have made contributions.

Authorization policy enforcement

Servers must enforce appropriate authorization policies on their clients to prevent unauthorized access to their internal resources. Many legacy servers, including window-management servers, game servers and middleware, either contain inadequate mechanisms or completely lack the ability to enforce authorization policies. My thesis research addresses the problem of retrofitting legacy servers for authorization policy enforcement.

Retrofitting servers for policy enforcement is currently an error-prone manual exercise. In particular, a developer faced with the task of adding authorization checks to such servers must answer two key questions: (1) what are the operations that a client can perform on a resource? and (2) how are these operations performed by the server? In current practice, these questions are answered manually by inspecting the server's source code. I have developed *program analysis and transformation techniques to reduce the manual effort required to retrofit legacy servers.* These techniques enable a developer to answer both questions (1) and (2), and add authorization checks in a principled way. As described below, the net result of these techniques is that the effort needed to retrofit large legacy servers reduces to from *several months* to just *a few hours* of manual effort.

The cornerstone of these techniques is a formalism called *fingerprints* that helps characterize how the server manipulates its internal resources [1]. Fingerprints are code-level signatures of the operations that clients can perform on resources. For example, the fingerprint "*Set* `Window/mapped` to *true*" for the X11 server matches statements that set the `mapped` field of a window to true. The X11 server executes a statement that matches this fingerprint each time it maps a visible window to the screen in response to an X client request. Much like a human fingerprint identifies an individual, this fingerprint identifies the operation of mapping a visible window to the screen. Once a set of fingerprints has been identified for a server, it can easily be transformed by adding authorization checks at each source code location that matches these fingerprints. I have developed both dynamic and static program analysis techniques to automate fingerprint-finding.

The dynamic technique [2] assumes that a high-level description of resource operations is available (*e.g.,* that mapping a window is an operation) and automatically finds fingerprints for these operations by analyzing execution traces of the server. Hence, this technique helps a developer answer question (2) above by finding code-level signatures for resource operations. The key observation used here was that operations on resources are typically associated with tangible side-effects on the resources. For example, when the map operation is performed on a window resource, a visible window appears on the screen. I devised a technique that used these side-effects as hints to determine whether an operation is performed on a resource during server execution and prune traces to find fingerprints. I implemented this technique in a tool called Aid. Experiments with Aid applied to the X11 server showed that it can drastically reduce the search-space for fingerprints. Each fingerprint was localized to within 15 functions, on average, in the X11 server; the alternative is to manually study the entire code-base of the X11 server. An X11 server instrumented by using these fingerprints to locate resource operations resisted previously-published attacks.

The static technique [4] for fingerprint-finding enhances the dynamic technique in two ways. First, it automatically mines a set of candidate resource operations, thus eliminating the need for *a priori* description of resource operations. Second it provides near-complete code coverage and finds all fingerprints: the dynamic technique cannot find all fingerprints. Hence, this technique helps a developer answer both questions (1) and (2). This technique automatically mines resource operations and their fingerprints by *clustering* resource manipulations. Using a technique called *concept analysis* for clustering, I enhanced Aid and applied it to find fingerprints in three legacy servers: the ext2 file system, the X11 server, and a game-server called PennMUSH. In each case, with just a few hours of manual effort, I was able to find resource operations (and their fingerprints) that were independently identified manually. For example, Aid's analysis of ext2 for directory operations produced 18 clusters, of which 11 corresponded to operations identified for security-enhanced Linux. Further, because the static technique provides near-complete code coverage and finds all fingerprints, I could directly verify that for the X11 server, all the fingerprints found by the dynamic technique were also found by the static technique.

Vulnerability and exploit detection

In collaboration with researchers at Grammatech Inc., I devised a technique for statically detecting buffer overflows in C source code [3]. The technique reduced the problem of detecting buffer overflows into finding the least fixed point of a system of linear constraints that was extracted from source code. A key contribution of this work was the observation that this fixed point could be computed efficiently using linear programming. I lead the development of a tool based upon this technique that used off-the-shelf linear program solvers to efficiently find several new vulnerabilities in widely-deployed software. For example, the tool found 14 new vulnerabilities in `wu-ftpd`, a popular FTP server.

Two observations from the above exercise motivated my next project: (1) vulnerability detection tools often produce false positives, and (2) not all vulnerabilities reported by such tools are exploitable. The main problem with vulnerability detection tools is that they typically do not model low-level program data, such as stack layout. These very details are used in real-world exploits, and it is important to model them to find exploitable vulnerabilities. Modeling and reasoning about low-level details requires the use of decision procedures for expressive logics.

In collaboration with researchers from Carnegie Mellon University, I formalized and solved the problem of finding exploits at the API-level [5]. As a case study, we modeled format-string exploits as API-level exploits, and built a tool to find such exploits. Exploits so found can be used to prioritize vulnerabilities to be fixed, and can potentially be used to find false positives reported by vulnerability detection tools: if no exploits are found against a vulnerability, it is likely a false positive. Our tool could find exploits against vulnerabilities in widely-deployed software (e.g., `php`, `qpopper`, `wu-ftpd`) within minutes and could also find false positives in reports produced by other vulnerability detection tools.

Future Agenda

In the near term, I plan to further my research agenda on securing legacy software along two directions—*ease-of-use* and *effective containment*.

Ease of use

A security system must be easy to configure and use. Unfortunately, today's security systems are complex and hard-to-configure, thus increasing the chance of operator error. As a result, they are prone to breaches. For example, correctly formulating authorization and information-flow policies in systems such as security-enhanced Linux is a challenging task even for an operator with significant domain expertise.

Attackers typically bypass authorization policy enforcement systems by exploiting inconsistencies between an authorization policy and security requirements. Can we ensure that an authorization policy satisfies a security requirement? If it does not, how should we update the policy so that it satisfies the requirement? To answer these questions, I plan to explore automated techniques for authorization policy analysis and update. One possible approach is to use model checking: a model checker either determines that the policy satisfies the security requirement, or emits a counterexample that violates the security requirement. The counterexample, which corresponds to an attack, can then guide an automatic tool to modify the policy and prevent the attack. This project bears close parallels to, and can therefore adapt and benefit from work on counterexample-guided verification.

A more ambitious effort is to develop a system that completely frees security analysts from the burden of writing authorization policies by automatically synthesizing these policies. The system would accept security requirements as input and would explore the space of authorization policies that satisfy the requirement. Important challenges here will be to design an expressive, yet usable language to specify security requirements and to devise techniques to constrain the search space of acceptable authorization policies. This project can benefit from research on program synthesis, where the goal is to automatically synthesize programs from specifications given as logic formulas.

Effective containment

In spite of several efforts to secure software, attackers innovate and devise novel strategies that breach security. Assuming that insecure software is an artifact to live with, can we refactor it so as to restrict the damage caused by attacks? I plan to investigate program transformation techniques to modify legacy software to contain the effect of attacks. The idea is to refactor software into modules that communicate via narrow, well-defined interfaces, and secure these interfaces so that a compromised module does not affect others. For example, one approach to achieve this is to run each module in its own protection domain (e.g., a lightweight virtual machine). Challenges to be addressed here include: (1) reducing the communication overhead between modules; (2) automatically identifying compromised modules and restricting their access to sensitive data; and (3) restoring compromised modules without affecting the functioning of other modules. Machinery developed to do so will also find potential applications in other areas. For example, it can be used to improve operating system reliability by rearchitecting device-drivers, errors in which are a major source of crashes today.

My longer term goal is to research effective ways to transform legacy software by equipping it with mechanisms to detect attacks, contain them and prevent future instances of these attacks—in short, make legacy software *security-aware*. Doing so without overly complicating the code or affecting its performance are challenges that I look forward to addressing in the future.

References

[1] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *ACM CCS'05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 330–339, Alexandria, Virginia, USA, November 2005. ACM Press.

[2] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE S&P'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229, Berkeley/Oakland, California, USA, May 2006. IEEE Computer Society Press.

[3] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *ACM CCS'03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354, Washington, DC, USA, October 2003. ACM Press.

[4] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. Mining security-sensitive operations in legacy code using concept analysis. In *ICSE'07: Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society Press.

[5] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. Automatic discovery of API-level exploits. In *ICSE'05: Proceedings of the 27th ACM/IEEE International Conference on Software Engineering*, pages 312–321, St. Louis, Missouri, USA, May 2005. ACM Press.

[6] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *ACSAC'02: Proceedings of the 18th Annual Computer Security Applications Conference (Compendium of Classic Papers)*, pages 119–126, Las Vegas, Nevada, USA, December 2002. IEEE Computer Society Press.