Dealing with System Monocultures

Angelos Keromytis
Computer Science Departmet
Columbia University
New York, NY, USA

angelos@cs.columbia.edu

Vassilis Prevelakis
Computer Science Department
Drexel University
Philadelphia, PA, USA

vp@cs.drexel.edu

ABSTRACT

Software systems often share common vulnerabilities that allow a single attack to compromise large numbers of machines (write once, exploit everywhere). Borrowing from biology, several researchers have proposed the introduction of artificial diversity in systems as a means for countering this phenomenon. The introduced differences affect the way code is constructed or executed, but retain the functionality of the original system. In this way, systems that exhibit the same functionality have unique characteristics that protect them from common mode attacks. Over the years, several such have been proposed. We examine some of the most significant techniques and draw conclusions on how they can be used to harden systems against attacks.

1. INTRODUCTION

The recent widespread disruptions of systems across the Internet underlined the inherent weakness of an infrastructure that relies on large numbers of effectively identical systems. Common elements in these systems include the operating system, the system architecture (e.g., Intel Pentium), particular applications (e.g., email, Word processing software), and the internal network architecture. Common-mode attacks occur when an attacker exploits vulnerabilities in one of these common elements to strike large numbers of victim machines. If each of these systems were different, then the attacker would have to customize their technique to the peculiarities of each system, thus reducing the scope of the attack and the rate of its spread.

However, running different systems in a network creates its own set of problems involving configuration, management and certification of each new platform. In certain cases, running such multi-platform environments can decrease the overall security of the network [1]. The premise of this paper is that by introducing randomness in existing systems we can vary their behavior sufficiently to prevent common mode attacks. Thus, our systems are similar enough to ease administration, but sufficiently different to resist common mode attacks.

Randomization can be introduced in various parts of a system. Areas include the configuration of the network infrastructure so that remote attackers cannot target a specific host or service (e.g., the White House site or the Microsoft software update server), the implementation of specific protocols (e.g., changing some aspects of the TCP/IP engine to reduce the risk of fingerprinting), or even the processor architecture to guard against foreign code injections attacks. In this paper, we describe various randomization techniques and examine how they can be used to strengthen the security of systems.

2. CLASSIFICATION

The diversification techniques that have been proposed over the years can be broadly classified into three categories: those that modify the structure of the system, those that modify the execution environment, and those that affect the system behavior. For example, systems such as StackGuard insert code that verifies the integrity of the stack every time the code returns from a subroutine call, whereas the Instruction-Set Randomization technique changes the instruction set of the processor so that unauthorized code will not run successfully.

2.1 Modifying the Structure

Structure modification techniques insert special code that performs sanity or consistency checks at various points in the execution of the program.

Perhaps the best-known of these techniques is StackGuard [2], a system that protects against buffer overflows. This is a patch to the popular gcc compiler that inserts a canary word right before the return address in a function's activation record on the stack (Figure 1). The canary is checked just before the function returns, and execution is halted if it is not the correct value, which would be the case if a stack-smashing attack had overwritten it. This protects against simple stack-based attacks, although some attacks were demonstrated against the original approach [3], which has since been amended to address the problem.

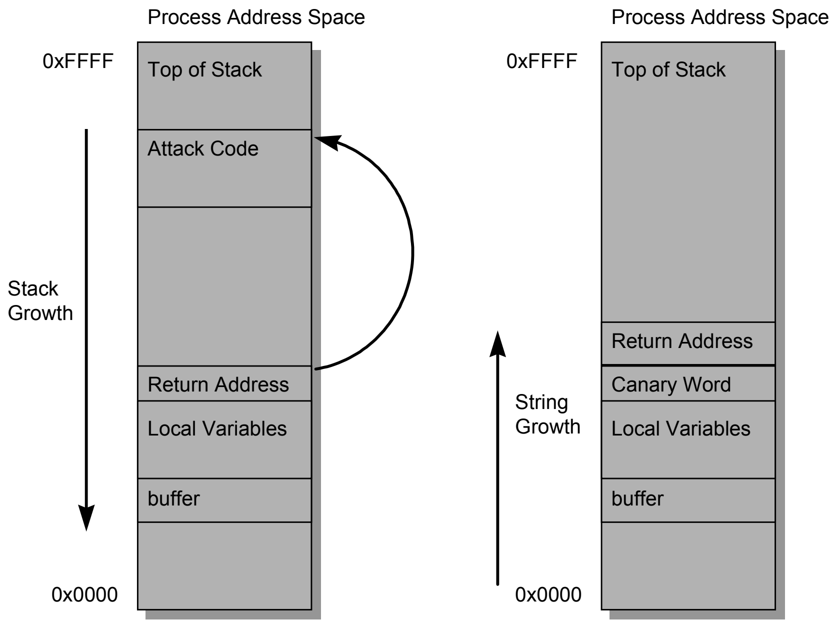

Figure 1: Stack buffer overflow (left) and StackGuard-modified stack frame (right).

Stack Guard is one of many similar systems such as MemGuard [2], FormatGuard [4], ProPolice [5], etc. Generally, these approaches have three limitations. First, the performance implications (at least for some of them) are non-trivial. Second, they do not seem to offer sufficient protection against stack-smashing attacks on their own, as shown in [3, 6] (although work-arounds exist against some of the attacks). Finally, they do not protect against other types of code-injection attacks, such as heap overflows [7]. For the purposes of our discussion, however, these techniques have the problem that they make deterministic changes to the code, and thus cannot protect against monoculture threats.

A system that is more applicable to this discussion is PointGuard [8] which encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. This is implemented as an extension to the gcc compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme. Another approach, address obfuscation [9], randomizes the absolute locations of all code and data, as well as the distances between different data items. Several transformations are used, such as randomizing the base addresses of memory regions (stack, heap, dynamically-linked libraries, routines, static data, etc.), permuting the order of variables/routines, and introducing random gaps between objects (e.g., randomly pad stack frames or malloc'ed regions). Although very effective against jump-into-libc attacks, it is less so against other common attacks, due to the fact that the amount of possible randomization is relatively small. However, address obfuscation can protect against attacks that aim to corrupt variables or other data.

A persistent concern in employing techniques such as the ones described above, is to maintain the efficiency of the application. In other words, the overheads associated with the use of these mechanisms must be minimized. Naturally, this discourages the use of more exhaustive and hence more expensive techniques. If, however, we can identify the parts of the code where a bug has a higher probability of resulting in a security vulnerability, we can reserve the use of the more expensive mechanisms to these sensitive regions.

Tools developed under the DARPA funded CHATS/CoSAK project facilitate the identification of such regions. This work is based on the assumption that a small percentage of functions near a source of input (such as file I/O), called *Inputs*, are the most likely to contain a security vulnerability [17]. The original hypothesis was confirmed by reviewing large numbers of bugs that have been posted in security forums such as the CERT. These reports also include the patches that correct the bugs, thus identifying the code that was responsible for the vulnerability (called *Targets*). The analysis of the existing systems revealed that Targets tend to be located "near" Inputs (where "near" is defined as a number of function calls). With this information, new systems can be analyzed by the CoSAK tools. The way they work is by examining the source code of a computer system in order to identify the Inputs. Then a call graph of the entire system is generated and the code appearing within a set number of function calls from the Inputs is pinpointed. Special mechanisms (e.g. code emulation, execution under a virtual environment, or limitations on privileges) can be activated when the flow of control strays into the sensitive regions.

2.2 Modifying the Environment

Systems do not exist in isolation but they need to interact with their environment (be it the processor architecture, the operating system, the network topology, etc.). To see how randomization techniques can be used to influence the execution environment let us look a bit closer at the problem of code-injection attacks.

Code-injection attacks attempt to deposit executable code (typically machine code, but there are cases where intermediate or interpreted code has been used) within the address space of the victim process, and then pass control to this code. These attacks can only succeed if the injected code is compatible with the execution environment. For example, injecting x86 machine code to a process running on a SUN/SPARC system may crash the process (either by causing the CPU to execute an illegal op-code, or through an illegal memory reference), but will not cause a security breach. Notice that in this example, there may well exist sequences of bytes that will crash on neither processor.

The instruction randomization technique [11] leverages this observation by creating an execution environment that is unique to the running process, so that the attacker does not know the "language" used and hence cannot "speak" to the machine. This is achieved by applying a reversible transformation between the processor and main memory. Effectively, new instruction sets are created for each process executing within the same system. Code-injection attacks against this system are unlikely to succeed as the attacker cannot guess the transformation that has been applied to the currently executing process. Of course, if the attackers had access to the machine and the randomized binaries through other means, they could easily mount a dictionary or known-plaintext attack against the transformation and thus "learn the language". However, we are primarily concerned with attacks against remote services (e.g., http, dhcp, DNS, and so on). Vulnerabilities in this type of server allow external attacks (i.e., attacks that do not require a local account on the target system), and thus enable large-scale (automated) exploitation. Protecting against internal users is a much more difficult problem, which we do not address in this work.

The power of the technique can be demonstrated by its applicability to other settings, such as SQL injection attacks. Such attacks target databases that are accessible through a web front-end, and take advantage of flaws in the input validation logic of Web components such as CGI scripts. The concept of instruction randomization has been applied to that setting, to create instances of the SQL language that are unpredictable to the attacker. Preliminary results indicate that the mechanism imposes negligible performance overhead to query processing, and can be easily retrofitted to existing systems. The same technique can easily be applied to any interpreted-language setting that is susceptible to code injection attacks.

In a different context, randomization of a system's environment has been used to combat network-based denial of service (DoS) attacks. The Secure Overlay Services (SOS) [18] approach addresses the problem of securing communication on top of today's existing IP infrastructure from DoS attacks, where the communication is between a predetermined location and users, located anywhere in the wide-area network, who have authorization to communicate with that location. The scheme was later extended to support unknown users, by using Graphic Turing Tests to discriminate between zombie machines and real humans [19].

In a nutshell, the portion of the network immediately surrounding the target (location to be protected) aggressively filters and blocks all incoming packets whose source addresses are not "approved". The small set of source addresses that are "approved" at any particular time is kept secret so that attackers cannot use them to pass through the filter. These addresses are picked from among those within a distributed set of nodes throughout the wide area network, that form a secure overlay: any transmissions that wish to traverse the overlay must first be validated at entry points of the overlay. Once inside the overlay, the traffic is tunneled securely for several hops along the overlay to the "approved" (and secret from attackers) locations, which can then forward the validated traffic through the filtering routers to the target. The two main principles behind this design are: (i) elimination of communication "pinch" points, which constitute attractive DoS targets, via a combination of filtering and overlay routing to obscure the identities of the sites whose traffic is permitted to pass through the filter, and (ii) the ability to recover from random or induced failures within the forwarding infrastructure or among the overlay nodes.

The overlays are secure with high probability, given attackers who have a large but finite set of resources to perform the attacks. The attackers also know the IP addresses of the nodes that participate in the overlay and of the target that is to be protected, as well as the details of the operation of protocols used to perform the forwarding. However, the assumption is that the attacker does not have unobstructed access to the network core. That is, the model allows for the attacker to take over an arbitrary (but finite) number of hosts, but only a small number of core routers. It is more difficult (but not impossible) to take control of a router than an end-host or server, due to the limited number of potentially exploitable services offered by the former. While routers offer very attractive targets to hackers, there have been very few confirmed cases where take-over attacks have been successful. Finally, SOS assumes that the attacker cannot acquire sufficient resources to severely disrupt large portions of the backbone itself (i.e., such that all paths to the target are congested).

Under these assumptions, by periodically selecting a new "approved" overlay node at random, a site can allow only authorized clients to communicate with it. An attacker must either amass enough resources to subvert the infrastructure itself, or attempt to guess the identity of the current approved node. Effectively, SOS allows the creation and use of an arbitrary number of virtual topologies over the real network (which can, perhaps perversely, viewed as a monoculture), which only legitimate users can use. The performance impact of doing so is studied in [19]. To summarize, end-to-end latency is increased by a factor of 2, while remaining impervious to the effects of a DoS attack.

2.3 Modifying the Behavior

Computer systems are to a large extent deterministic and this can be used as a means of identification (fingerprinting), or, worse, as means of subverting a system by anticipating its response to various events.

Fingerprinting is a technique that allows remote attackers to gather enough information about a system so that they can determine its type and software configuration (version of operating system, applications etc.) [14]. This information can then be used to determine what vulnerabilities may be present in that configuration and thus better plan an attack.

Having a system with predictable behavior can have devastating consequences for its security. The most celebrated example is the attack that exploited easy to guess TCP/IP packet sequence numbers [15]. By being able to guess the sequence number of a TCP connection with a remote system, we can construct and transmit replies to packets that we never receive (perhaps because a firewall prevents the remote system from talking to us, or because we use a spoofed source address in our packets).

More recently, a denial of service attack based on the TCP retransmission time-out [16], allowed an attacker to periodically send bursts of packets to the victim host, forcing the TCP subsystem on the victim host to repeatedly time-out causing near-zero throughput. In this case as well, by changing the behavior of the TCP implementation (randomizing the retransmission time-out), the attack can be mitigated.

The general Internet philosophy of "being conservative in what you send and liberal in what you accept" (RFC1341), while enhancing interoperability, sometimes creates vulnerabilities by allowing greater ambiguity in what a networked application may accept. Especially in the case of the Internet Protocols these minor variations have been used as the basis of attacks (e.g. the overlapping fragment attacks and the small packet attacks of the early 90s), and more recently as a means to facilitate fingerprinting.

OpenBSD's packet filter, pf (4), includes a "scrub" function that normalizes and defragments incoming packets. This allows applications and hosts on the internal network some form of protection against hand-crafted packets designed to trigger vulnerabilities. Another approach is to apply a similar technique to outgoing packets in order to hide identifying features of the IP stack implementation [20]. A key part of the process of the obfuscation process is protection against time-dependent probes. Different TCP implementations have variations in their time-out counters, congestion avoidance algorithms, etc. By monitoring the response of the host under inspection to simulated packet loss, the timing probe can determine the version of the TCP implementation and by extension that of the OS. Also the use of various techniques for rate limiting ICMP messages by the victim system, can provide hints to the attacker. The effectiveness of such probes can be reduced, by homogenizing the rate of ICMP traffic going through the system that connects the trusted network to the outside world, or by introducing random delays to ICMP replies.

3. SUMMARY AND CONCLUDING REMARKS

The commoditization of computer systems has dramatically lowered the cost of ownership of large collections of computers. It is thus no longer economically feasible to have one-off configurations for individual computers or networks, which, in turn, leads to monocultures, vulnerable to common-mode attacks. There is a lively debate going on as to the effects of a diverse computing environment on security. One camp claims that diversity is not required as it distracts from the task of producing a single secure configuration that can then be widely deployed, thus spreading the development and security administration costs to a large number of machines. The other camp claims that by standardizing the interfaces between subsystems, multiple implementations can be deployed, thus reducing the risk of a single problem affecting all the deployed systems. Our view is that both sides are fundamentally wrong. Having potentially huge numbers of identically configured hosts invites disaster: no amount of effort can secure large software systems that have not been built with security in mind. Even in cases where formal methods have been used in the design, field upgrades and maintenance can weaken the security posture. On the other hand, attempting to introduce diversity through the development of different software systems is not viable. Designing, developing and maintaining a system is so expensive that once we have a working version we tend to use it widely. Even in critical systems such as avionics, the same software is used on multiple hardware platforms (creating redundancy only at the hardware level). The failure of the inaugural flight of the Ariane 5 launcher due to a software bug crashing both navigation computers is proof that having the same software running on redundant hardware does not provide true redundancy.

Our intention has been to demonstrate that the *effects* of diversity can be introduced through automated means. The techniques described in this paper allow the introduction of small but critical variations to the these off-the-shelf systems. While randomization is by no means the silver bullet that will solve the problem of generic software, or system exploits (these can only begin to be addressed if we abandon the current ad hoc design and development techniques) they do provide an effective method for mitigating attacks and exposing the bugs that make such attacks possible.

4. REFERENCES

1. Prevelakis, V.: A secure station for network monitoring and control. In: Proceedings of the 8th USENIX Security Symposium. (1999)

2. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium. (1998)

3. Bulba, Kil3r: Bypassing StackGuard and StackShield. Phrack 5 (2000)

4. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In: Proceedings of the 10th USENIX Security Symposium. (2001) 191-199

5. Etoh, J.: GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/ (2000)

6. Wilander, J., Kamkar, M.: A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention. In: Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS). (2003) 123-130

7. M. Conover and w00w00 Security Team: w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt (1999)

8. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium. (2003) 91-104

9. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: Proceedings of the 12th USENIX Security Symposium. (2003) 105-120

10. DaCosta, D., Dahn, C., Mancoridis, S., Prevelakis, V.: Characterizing the Security Vulnerability Likelihood of Software Functions . In: Proceedings of the 2003 International Conference on Software Maintenance (ICSMy03). (2003) 61-72

11. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization. In: Proceedings of the ACM Computer and Communications Security (CCS) Conference. (2003)

12. Barrantes, G., Ackley, D., Palmer, T., Zovi, D.D., Forrest, S., Stefanovic, D.: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In: Proceedings of the ACM Computer and Communications Security (CCS) Conference. (2003)

13. Keromytis, A.D., Misra, V., Rubenstein, D.: Secure overlay services. In: Proceedings of the ACM SIGCOMM Conference. (2002) 61-72

14. Smart, M., Malan, R., Jahanian, F.: Defeating TCP/IP Stack Fingerprinting. In: Proceedings of the 9th USENIX Security Symposium. (2000) 229-240

15. Inc, S.N.: A simple TCP spoofing attack. http://niels.xtdnet.nl/papers/secnet-spoof.txt (1997)

16. Yang, G.: Low-rate denial-of-service (DoS) attacks to TCP. http://www.cs.ucla.edu/yangg/re-search/research.htm (2003)

17. DaCosta, D., Dahn, C., Mancoridis, S., Prevelakis, V.: Characterizing the Security Vulnerability Likelihood of Software Functions . In: Proceedings of the 2003 International Conference on Software Maintenance (ICSMy03). (2003) 61-72.

18. Keromytis, A.D., Misra, V., Rubenstein, D.: Secure overlay services. In: Proceedings of the ACM SIGCOMM Conference. (2002) 61-72.

19. Morein, W.G., Stavrou, A., Cook, D.L., Keromytis, A.D., Misra, V., Rubenstein, D.: Using Graphic Turing Tests to Counter Automated DDoS Attacks Against Web Servers. In: Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS). (2003) 8-19.

20. Smart, M., Malan, R., Jahanian, F.: Defeating TCP/IP Stack Fingerprinting. In: Proceedings of the 9th USENIX Security Symposium. (2000) 229-240