Server Protection through Dynamic Patching

Nicolas Loriant Marc Ségura-Devillechaise Jean-Marc Menaud
Obasco Group
EMN-INRIA, LINA
4, rue Alfred Kastler
44307 Nantes Cedex3, France
nloriant,msegura,jmenaud@emn.fr

Abstract

Recently, hackers has been developing fast propagating worms exploiting vulnerabilities that had just been disclosed by security experts. Those attacks particularly expose servers: this class of applications is constantly connected to the Internet and must meet uptime constraints. Hence they often run unprotected until the next scheduled update.

In this paper, we propose a just-in-time protection for servers based on runtime injection of pre-made patches. The runtime injection permits to deal with uptime constraints and induces only a minimal overhead over the vulnerable code and only when a vulnerability is known to exist. The pre-made patches forbid exploitation of most common vulnerabilities (45% of attacks reported by Debian security in 2005 affecting C softwares) and allows continuous servicing.

1. Introduction

Hackers no longer try to find unknown bugs, but instead attempt to exploit the latency between bugs disclosure and correction. Indeed, analysis of recent worms such as Sasser and Witty [12], showed that their creators exploited vulnerabilities that had just been published by security experts. This kind of attack is particularly problematic for applications such as `wu-ftpd` and `Apache` that must meet both uptime and performance constraints. Therefore, on bug disclosure such applications must run unprotected until the next scheduled update.

To protect applications against attacks while meeting uptime and performance constraints, we propose to correct vulnerabilities at runtime as soon as these are discovered. This approach requires both a technology to apply the update and a mean of ensuring that the new code does not interact badly with the current application state. As an integration technology, we propose to use Arachne, a runtime binary rewriting tool [3]. The contribution of this paper lies in how we address the verification issue. We have observed that most attacks exploit only three kinds of vulnerabilities: buffer overflows, double free bugs, and format string bugs. Our contribution resides hence in a set of three generic, premade patches each correcting one of these vulnerabilities without disturbing normal execution and with low performance loss. Our patches are applicable at runtime on any C written application on an IA-32 Linux platform. Using our approach, we may protect against 45% of attacks reported by Debian security in 2005 over C softwares.

Our patches has to be configured by system administrators to properly locate bugs (*e.g.* the buffer name for a buffer overflow) and to define the action taken on attack detection. The location information is used to protect only the vulnerable code and thus limit the cost of the protection mechanism. The action can be either to abort the application or to choose an appropriate behavior (*e.g.* to ignore out of bounds data on a buffer overflow or to resize the vulnerable buffer) and to continue. Hence it permits not only to detect attacks but also to correct vulnerabilities at runtime. Once the patch is configured, it is injected into the program at runtime through the Arachne binary rewriting tool. As the protection is injected only when a vulnerability is known to exist, the cost induced by the protection is only paid when necessary. Furthermore, the dynamic injection and resuming capability of our patches respect uptime constraints of servers.

The rest of this paper is organized as follow, Section 2 starts with an overview of buffer overflows, double free bugs and format string bugs, followed by the workflow we propose to correct vulnerabilities at runtime. In Section 3, we show how our patches are configured by the system administrator, how they are injected dynamically into the running application, and how they detect and protect against exploit attempts. In Section 4, we evaluate our proposal in

terms of ease of use, performance and security coverage. Section 5 discusses related work and Section 6 concludes.

2. Dynamic Patching

This section first briefly presents the vulnerabilities we target, and then describes the workflow followed by system administrators to correct vulnerabilities at runtime.

2.1. Vulnerability description

Binary injection attacks require modifying data in the process' memory, whether control data (*e.g.* the function return address) in order to execute malicious code, or non-control data (*e.g.* user permissions associated with a file descriptor). Buffer overflows, double free bugs and format string bugs are three common vulnerabilities that can be exploited in order to write arbitrary values at arbitrary locations or to read sensitive data.

Buffer overflow is the most common vulnerability exploited by hackers and have been extensively analyzed [5]. Buffers, whatever their allocation mechanism is (static, dynamic, heap, stack), may overflow when no check ensures the data written into it actually fits in. When a buffer overflows, data structures adjacent to it in memory are overwritten by extra data. This is often exploited by hackers in order to deviate the target program's control flow to a specific location, *e.g.* a shell code.

Double free bugs appear when a not previously allocated memory block is freed. The GNU C library does not check the proper allocation before freeing a memory block. When a hacker can force the program to free a buffer twice and to reallocate it, he can trick the GNU C library into rewriting a given location with an arbitrary value. This bug is not proper to the GNU C library. Other standard library implementations suffer from the double free vulnerability, including system V AT&T implementation and Microsoft Windows implementation, namely RtlHeap.

Format string bugs Functions in the `printf` family use a format string made of wildcards to describe the remaining arguments. When a `printf`-like function uses user input as the format string, a hacker can force the function to use arguments that are not present. As arguments are passed via the stack, the program can be tricked into treating improper stack values as parameters. Typically, the `%n` directive, which stores the number of characters written at the address given in the corresponding argument permits to rewrite an arbitrary word at an arbitrary place. Format string bugs are not proper to `printf`-like functions only, but affect all variadic functions: functions that accept variable number of arguments, as neither the C language nor the execution environment ensure proper consumption (number and type) of arguments passed to variadic functions.

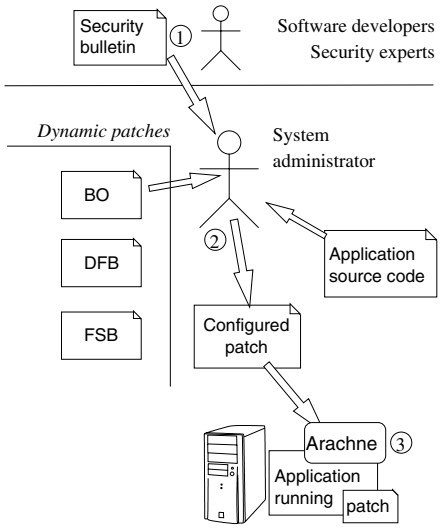

Figure 1. Workflow of dynamic patching

2.2. Workflow

In this section, we describe the workflow we propose to adopt to correct vulnerabilities at runtime. When a vulnerability is discovered in an application by developers, security experts or external contributors, a security alert is published through mailing list or security centers (Figure 1, ①). This bulletin provides information such as the application affected, the vulnerability type, its severity, and others. It may also contains instructions to isolate the problem or a patch.

When the system administrator receive this security bulletin ②, in addition to determine if the vulnerability concerns a software installed on his network, the system administrator has to find first if the software affected is C written and second if the vulnerability is a Buffer Overflow, a Double Free, or a Format String. Then according to the vulnerability's type, the system administrator has to retrieve the bug location from the security bulletin, the application and a patch optionally provided (in Section 2.1 we describe the information required). Then the system administrator may choose between different types of actions to be taken upon the detection of an exploit attempt. In Section 4.1, we show this whole step can be done in about 8 minutes.

Once the patch has been configured, it is dynamically injected into the running application ③. For this, we use Arachne, our binary rewriting API descibed in Section 3.2.

3. Implementation

In this section, we first describe the patch configuration process. We then present the techniques used to support runtime patch injection. And finally, we explain how our patches handle to detect and protect against exploit attempts.

3.1. Patch configuration

The system administrator must provide information allowing our tool to locate and to correct only the vulnerability targeted. In most cases, this information is enclosed in the security bulletin. In the case of a buffer overflow, our tool needs the name of the buffer. For the correction of a format string bug, the tool requires the name of the vulnerable variadic function. Only double free bugs do not require any user-provided information as our patch corrects every potential double free bug. Additionally, to further reduce the protection overhead, the user may provide names of functions in which buffers are accessed in an illegal way, or illegal calls are made to a variadic function or to `free`.

Once the system administrator has described the vulnerability to be corrected, he can choose between various behaviors carried out upon the detection of an exploitation attempt. Even if most of the literature advise to abort the program execution upon an attack, Rinard and *al.* have pointed out that in certain cases it is preferable to continue the program execution after prohibiting the attempt [11]. Our tool allows the user to select one of these two solutions. When resuming is chosen, the behavior depends on the vulnerability type. On an invalid read access attempt caused by a buffer overflow or a format string bug, the patch returns a dummy or user-specified value. Invalid writes for Buffer Overflows and Format String Bugs or invalid calls to `free` for double free bugs are simply skipped.

3.2. Patch injection

Once configured, patches are merged with the running application through the binary rewriting API of Arachne, a prototype developed in our previous work on runtime software modifications [3]. Arachne is independent of both compiler and runtime environment and is thus applicable to any ABI compliant software running on any x86 machine. As an instrumentation back-end of our just-in-time protection, Arachne's API is responsible of loading patches in the address space of the application prior its instrumentation and handling all the nitty-gritty of the binary manipulations. These operations should be carried out without suspending the execution of the application and in a reliable fashion.

Patches are compiled into native shared dynamically linked libraries. Arachne's API performs the patch injection in two steps. First, the Arachne instrumentation kernel is injected into the application's address space through the regular Unix debugging API (`ptrace`). This API imposes to suspend the application during the kernel injection: in practice, the duration of this suspension is about 100 ms on a Pentium 4. Once the kernel is loaded, it creates a thread in the application process, and waits for patch injection requests.

The code dynamically modified depends on the targeted vulnerability. The patch handling buffer overflows checks the size of any data written in dynamically allocated memory: it instructs Arachne to rewrite the standard function call (*i.e.* `malloc`) used to allocate memory. The patch handling double free bugs rewrites calls that dynamically allocate (*i.e.* `malloc`) and liberate (*i.e.* `free`) memory. The patch handling format string bugs instruments the invocations of the standard functions using format string arguments (*i.e.* `printf`, `fprintf`, `sprintf` and `snprintf`).

3.3. Exploit attempt detection

Buffer overflows Preventing a buffer overflow requires ensuring that there is no update modifying a location outside the boundary of the buffer. To achieve this independently of the target application, our pre-made patch reallocates the vulnerable buffer into a protected memory page. The application is then dynamically rewritten, so that it references the new location of the buffer. As the used memory page is read and write protected, every access to it triggers a "invalid memory reference" signal, `SIGSEGV`. This signal is trapped by the patch that compares the accessed location with the buffer limits before modifying the original data.

Our patch stores the location of an eventual signal handler registered by the application. When a `SIGSEGV` signal is triggered, our patch tests the source of the memory violation to determine if it has to be handled by the patch's signal handler or the original one. In addition, future calls to signal handler registering routines are monitored to enable base program to act normally.

Double free bugs The prevention from a double free bug requires ensuring that no block is freed when it was not actually allocated. Once applied, our pre-made patch monitors every call to allocation and free functions, and traces the allocated and freed blocks. On each `free`, the patch checks whether the call is legal. This check can be limited to calls specified by the system administrator to limit the overhead.

Here, our patch acts as an additional layer between the program and the memory allocation routines. Thus, it is independent of the implementation of these routines (GNU C library, System V AT&T...).

Format string bugs Preventing format string bugs requires ensuring a variadic function only accesses the provided parameters. On call of a variadic function, our patch checks the actual number of arguments passed. It also rewrites the body of the variadic function by replacing every parameter access with an `int3` instruction that triggers a signal `SIGTRAP`. The original instructions are thereby relocated in the signal handler. Then for each access, it compares the location accessed with the actual location of the parameter.

As previously described for buffer overflow patch, an eventual signal handler registered for `SIGTRAP` is preserved, to permit the program to act as would normally do.

4. Evaluation

To be useful our proposal must meet three goals. First it must be easily applicable by system administrators. Second it has to be efficient enough for the application to continue its task with a minimum overhead. Third it must be effective in detecting and forbidding attacks. For the first goal, we present statistics computed on 2005's security vulnerabilities found in open source softwares. We analyzed each bulletins in order to evaluate the time needed to find information about the security vulnerability. We show that for more than 60% of the vulnerabilities our patches target, the information required can be gathered manually in approximatively 8 minutes. To experiment with the second goal, we evaluated the overhead induced by our patches. We first present micro-benchmarks of our solutions for each of the bugs we handle. Then, we show macro-benchmarks that compare our proposal applied to the well-known ftp server `wu-ftpd` to its unprotected version. This shows that even if our solution might seem locally expensive, this overhead is negligible in large applications, making it suitable for real world applications. To assess the last goal, we discuss security coverage issues for each of our patches.

4.1. Approach feasibility

To evaluate the time invested by system administrators on bug disclosure to configure our dynamic patches, we computed statistics on security vulnerabilities found in open source softwares. We have chosen Debian security as a source of information as it covers a wide range of open source softwares. Debian reported 882 security bulletins

used `cscope`, a C cross reference tool to navigate application source code. We considered that system administrators would not devote more than 20 minutes per patch and then considered it as a hard limit for information extraction.

In over 20% of the alerts examined, there were no or irrelevant information in the bulletins, hence it was impossible to identify the vulnerability location. In such situation our proposal can not be used. Nevertheless we argue that such bulletins are no help either to hackers looking for vulnerabilities. Hence, there are fewer chances that an exploit will be developed quickly after the security alert. For 16% of the vulnerabilities, bulletins supply a source file name where the vulnerability is located. These cases are also problematic as it is quiet difficult to find relevant information rapidly, indeed source files are sometimes, like in `sendmail`, larger than 8000 lines of code. We were able to do it only for format string bugs because variadic functions are easy to find even in huge source files. Again we argue that this situation will not permit a hacker to exploit a buffer overflow after its disclosure.

For the remaining bulletins (63%), the given starting point was a function name. Because of different coding styles, function length may vary from few lines to hundreds of. For all these bulletins except one, we were able to identify the vulnerabilities in about 8 minutes. Figure 4 summarizes our success/failure in attempting to locate vulnerabilities according to information provided by security bulletins and no source patch were provided.

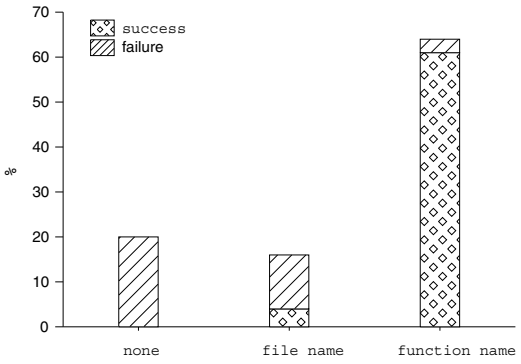

Figure 4. Success to locate vulnerabilities depending on information provided

The statistics we conducted clearly assess our solution can help in protecting applications against real vulnerabilities. Indeed, our solutions is practicable in approximatively 65% of buffer overflows, double free bugs and format string bugs in few minutes even when a patch does not exist yet. Moreover, we argue that when a patch is actually present, our solution is even quicker to apply as patches directly points relevant information.

4.2. Micro-benchmarks

To estimate the cost for each patch we implemented, we compared the cost of a normal execution with the one of the protected version of the application.

We had to address several problems to obtain fair measurements: *(i)* Pentium processors can reorder executed instructions, *(ii)* execution time of an instruction is dependent on the preceding one and *(iii)* experiments can be preempted during measurement. To avoid these problems, we *(i)* used serializing instructions that force the processor to process preceding instructions before continuation (these serialization instructions are not part of the measure), *(ii)* measured thousands of repetitions of the concerning execution at once and *(iii)* used the `getrusage` function to drop measures when the process had been preempted. Compared to Intel specifications that state an `nop` average instruction duration of 1 cycle, our experimental protocol permits to obtain a relative error of only 1.6%. Table 1 summarizes our experimental results.

Using Arachne in order to protect a buffer from an overflow implies that each access made to this buffer costs 9729 cycles instead of 1 cycle in its native version. This cost is easily explained by the fact that our patch introduces signals which imply context switches between kernel space and user space.

In order to protect an application against double free bug exploitation, our solution tracks memory allocation and deallocation, to detect and forbid freeing of already non-allocated memory. The gnu C library performs a call to `malloc` in 72 cycles and a call to `free` in 66 cycles. Our protected implementation extends `malloc` and `free` to monitor allocated and freed memory block using a hash table. This costs 5.1 times, and 5.7 times respectively, more than the gnu C library implementation.

Our solution for format string bugs relies on checking every parameter access made by a variadic function to validate that the requested location is actually that of a parameter. Each protected access is 2762 times slower than its native version. Again the reason for this, is the use of signals to trap accesses.

4.3. Macro-benchmarks

We also experimented with our patches on the real world application `wu-ftpd` (Washington University File Transfer Protocol Daemon), a widely deployed File Transfer Protocol service. It is a base for development of other ftp servers, *e.g.* BSD ftpd, ProFTPD, and then it is relevant of real application code.

	processor cycles		
	Safe	Native	Ratio
`buffer read`	9729	1	9729
`malloc`	372	72	5.1
`free`	378	66	5.7
`param read`	2762	1	2762

Table 1. Micro-benchmarks on patch protection mechanisms.

We performed our measures for a patched version of wu-ftpd, we chose to correct a buffer overflow vulnerability identified in the s/key authentication mechanism discovered in 2004 and referenced by the Common Vulnerabilities and Exposures under the identifier `CVE-2004-0185`. This vulnerability considered highly critical, permits a remote attacker to gain administrator access or to cause a Denial of Service.

In order to evaluate `wu-ftpd`'s performance, we used `dkftpbench` [6], a benchmarking tool for FTP servers. `dkftpbench` permits to stress ftp servers by faking client's connections through automatons. Each false client authenticates to the server, retrieve a given file, and disconnect. `dkftpbench` constantly create new automatons and then permits to measure instant/average/max number of simultaneous users. We performed our measure between 2 machines: one for `dkftpbench` and one for `wu-ftpd` over a 100 Mb/s Ethernet. `wu-ftpd` was running on a Pentium 4 3.3 GHz with 512 MB RAM. Each file retrieved was 5 MB long and network bandwidth dedicated to each client was set so that network never reached congestion.

We measured the maximum simultaneous users served by `wu-ftpd` when running unprotected and protected. Results shows no significant difference between the two versions 1008, and respectively 1012 simultaneous users. This demonstrates that even if our patches might seem locally resources consuming, they are clearly suitable in real world situations by permitting to protect applications against attacks without impacting significantly on performance.

4.4. Security coverage

For dynamically allocated buffers, the size is only known at allocation-time. Thus our patch can only protect dynamically allocated buffers that have been allocated after the patch is injected. Similarly, double free bugs cannot be forbidden if the block allocation occurred before the patch injection. Furthermore, if the first event caught is a `free`, there is no way to know if it is legal or not. When attacking servers, data exploited are often short lived data as they correspond to requests, thus even if our solution might not protect some data during a short period after injection, it becomes effective quickly when a new request is received.

Our patch for format string bugs ensures that every access to parameters will reach a parameter. This fails under certain circumstances to protect against format string attacks. Our patch won't check for type correctness of the parameters. Even if an exploitation is still theoretical possible, this is very difficult as the hacker would have to pass both specially crafted format string and parameters.

5. Related work

To our knowledge, no security solution addresses both uptime and performance constraints. For example, the usual patching process with the `patch` tool requires a patch and to stop the application. To permit patching without requiring to stop the service provided, hardware replication might be used [10]. This solution implies additional investment and maintenance load. Moreover it may requires complex techniques to duplicate/transfer current state and work in progress between machines.

Compiler extensions techniques [2, 5] detect both buffer overflows and format string bugs by inserting additional source code prior to compilation to enforce checks on buffer boundaries and variadic functions' usage. Because these solutions attempt to protect all buffer to overflow or all potential format string miss-usage, they introduce an important overhead of up to 400% depending on the application usage of buffers and variadic functions. As our solution focuses on bugs that are likely to be exploited, its consumes fewer resources.

The GNU C library proposes a environment variable `MALLOC_CHECK_` to set up the memory allocation mechanisms to correct at runtime memory corruption due to double free bugs. Most general Linux distributions deactivate this option as it severely impacts on performance. Our patch for double free bugs contrary to the GNU C library protection can be activated/deactivated at runtime, and thus permits to better deal with performance issues.

The PaX Project proposes two mechanisms to harden bugs exploitation [9]. First, non-executable protection that makes heap and stack segments non executable permits to protect against shell code execution. Nevertheless the new attack strategy "return-into-lib(c)" permitted to defeat non-executable pages [13]. As "return-into-lib(c)" attacks relies on the knowledge of functions address, the second mechanism proposed, ASLR, Address Space Layout Randomization adds a new order of difficulty for hackers. Nevertheless, again hackers have shown that ASLR mechanism was not generating enough entropy to effectively protect against brute force attacks [8]

Chen and al. propose a hardware based solution to tackle binary injection attacks [1]. They propose to track values provided through user inputs and to forbid dereferencing of

such values. This solution induces an important memory loss (a bit is associated to every byte in memory) and consider a value to be sanitized once any check has been performed on it. This strategy is arguable as it will not detect attacks through user inputs with but not enough, sanitizing.

Modern IDEs such as Eclipse and IntelliJ IDEA propose an "Edit and Continue" feature based on Java HotSwap [14]. While this approach permits runtime code patching, debugging options required at load-time makes it less suitable to cope with the unanticipated nature of bugs. The C/C++ interpreter CINT [4], also permits runtime code modification, but does not fully support the C language and is not multi-thread safe, hence it might not suit well to real world applications.

We previously proposed a solution based on source patch transformation. We built Minerve a source patch to aspect transformer [7]. Used in conjonction with Arachne, Minerve permits to automaticaly and dynamicaly apply source patch at runtime. While this solution copes with both uptime and security issues, it suffers from few drawbacks. First, it always requires a patch to be provided. Second, it is hard even sometimes impossible to ensure a patch developed with no dynamic issues in mind, may be applied at runtime.

6. Conclusion

In this article we have proposed a solution to correct common vulnerabilities in C written applications. In the context of highly available servers, our approach proves as a well-suited solution and ensures service continuity. With our three pre-made patches for the just-in-time correction, we cover 45% of the attacks affecting C softwares reported by Debian security in 2005.

Being transparent for the corrected process, our solution does not impose any modification on the compilation or execution environment. Moreover, the configurability of our pre-made patches lets the system administrator define the action carried out on an attack, and limit the overhead of the protection mechanism.

We show the configuration made by the system administrator could be made with few effort (approximatively 8 minutes per patch). Nevertheless, our solution would clearly benefits from its support by a central authority and developers that would relieve system administrators from the largest part of the configuration process.

After having evaluated our approach on the `wu-ftpd` server, we now intend to investigate the performance on further server applications. Preliminary tests on the Web cache Squid have already confirmed the suitability of our solution. In addition, we will work on the extension of our approach to tackle the remaining 55% of reported vulnerabilities, noticeably race conditions.

Acknowledgements

This work is supported by a regional grant from the Pays de la Loire (France). The authors would like to thank Julia Lawall and Thomas Fritz for their valuable comments.

References

[1] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 378–387, Yokohama, Japan, June 2005. IEEE Computer Society.

[2] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. In *Proceedings of the Tenth Usenix Security Symposium*, Washington, DC, USA, Aug. 2001. USENIX.

[3] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD)*, pages 27–38, Chicago, IL, USA, Mar. 2005. ACM Press.

[4] M. Goto. CINT, The C/C++ Interpreter. `http://root.cern.ch/root/Cint.html`.

[5] R. W. M. Jones and P. H. J. Kelly. Backward-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 13–26, Linköping, Sweden, May 1997.

[6] D. Kegel. dkftpbench. `http://www.kegel.com/dkftpbench/`.

[7] N. Loriant, M. Ségura-Devillechaise, and J.-M. Menaud. Software Security Patches – Audit, Deployment and Hot Update. In *Proceedings of the Fourth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 25–29, Chicago, IL, USA, Mar. 2005.

[8] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), Dec 2001.

[9] PaX Project. Address space layout randomization, Mar 2003. `http://pageexec.virtualave.net/docs/aslr.txt`.

[10] D. Pescovitz. Monster in a box. *Wired*, 8(12), Dec. 2000.

[11] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee Jr. Enchancing Server Availability and Security through Failure-Oblivious Computing. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 303–316, San Francisco, CA, Dec. 2004. USENIX.

[12] C. Shannon and D. Moore. The spread of the witty worm. *IEEE Security & Privacy*, 2(4):46–50, July/Aug. 2004.

[13] Solar Designer. Getting around non-executable stack (and fix). *Bugtraq Mailing List*, Aug 1997.

[14] Sun Microsystems Inc. The Java HotSpot™ Technology. `http://java.sun.com/products/hotspot/`.