



New Security Enhancements in Red Hat Enterprise Linux v.3, update 3

By Arjan van de Ven

Abstract

This whitepaper describes the new security features that have been added to update 3 of Red Hat Enterprise Linux v.3: ExecShield and support for NX technology.

August 2004

Table of Contents

Introduction	2
Types of Security Holes	2
Buffer Overflows	3
Countering Buffer Overflows	4
Randomization	6
Remaining Randomization: PIE Binaries	7
Compatibility	8
How Well Does it Work?	9
Future Work	10



Introduction

The world of computer security has changed dramatically in the last few years. Network security used to be about one dedicated hacker trying to get into one government computer, but now it is often about automated mass attacks. The SQL Slammer and Code Red worms were the first wide-scale computer security incidents to get mainstream press coverage. Linux has had similar, less-invasive worms in the past, such as the Slapper worm of 2002.

Another relatively new phenomenon is that compromised computers are primarily being used for other purposes, including sending spam or participating in Distributed Denial of Service (DDOS) attacks.

A contributing factor to the mass-compromise problem is that a large portion¹ of users and system administrators generally do not apply the security fixes that are provided by the operating system vendor. This leaves a significant number of vulnerable machines connected to the Internet at all times.

Providing security updates after the fact, however, is not sufficient. Operating system providers need to be more proactive in combating security problems.

This paper describes the ExecShield technology that Red Hat developed and included in update 3 of Red Hat Enterprise Linux v.3. This technology makes automated security exploits by malicious individuals more difficult or even impossible, thus reducing the impact of security vulnerabilities.

Types of Security Holes

There are numerous types of security vulnerabilities which can lead to problems. Security problems can be separated into two classes: social engineering and programming defects.

Social engineering vulnerabilities are best countered by a combination of user education and simple technical measures, such as making sure anti-virus software is installed and updated regularly. Viruses and *phishing*² are the most widespread forms of social engineering.

The most well-known programming defects are *buffer overflows*. Buffer overflows are common mistakes found in programs written in the C or C++ programming languages and are generally very easy to exploit. In fact, there are semi-automated exploit-creation kits available on the Internet.

In 2002, fixes for buffer overflows comprised about 22.5% of the security fixes provided by vendors³, while the percentage based on availability of exploits is much higher (about 75% over the last year).

¹ For example, refer to Eric Rescorla's paper for the 12th USENIX Security Symposium: http://www.rtfm.com/upgrade.pdf 2 Phishing attacks use spoofed e-mails and fraudulent websites designed to fool recipients into divulging personal financial data such as credit card numbers, account usernames and passwords, social security numbers, etc.

³ Based on data from: http://cve.mitre.org/board/archives/2002-10/msg00005.html



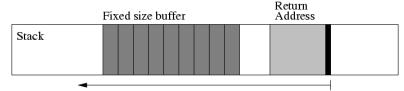
Buffer overflows are not the only type of programming flaw that lead to security breaches. Problems such as cross-site scripting, temp-file races, format-string and flawed permission checks can also lead to vulnerable systems. It is beyond the scope of this text to go into detail on these types of exploits.

The rest of this whitepaper will focus on buffer overflows and ways to limit the damage they can do.

Buffer Overflows

In order to explain what Red Hat has done to counter buffer overflow exploits, we must first explain how a typical buffer overflow exploit operates.

Figure 1. Typical stack layout



The stack frame image above shows the memory layout of a typical function (or subroutine) that uses a fixed size buffer. This buffer is stored on the stack and is located before the memory buffer containing the address of the the program code that invoked the subroutine. When the subroutine is finished, this address is used to resume the program at the point of the subroutine invocation.

On Intel and compatible processors,⁴ the stack grows in a downward direction over time (as seen at left in the stack frame image). This is why the buffer is stored before the return address.

A buffer overflow exploit operates by virtue of a defect in an application. For example, an exploit can trick a subroutine to put more data into the buffer than there is space available. This surplus of data is stored beyond the fixed size buffer, including the memory location that has the return address stored. By overwriting the return address (which holds the address of the memory location of the code to execute when the subroutine is complete), the exploit has the ability to control which code is executed when the subroutine finishes. The simplest and most common approach is to make the return address point back to the buffer.

⁴ Some other, less frequently used, architectures use different mechanisms than the one in the diagram.

Figure 2. Making the return address point to inside the buffer





After all, the exploit has just filled the buffer. Typical exploits fill the buffer with program code to be executed; code that may execute another program or otherwise damage or compromise the machine.

Countering Buffer Overflows

As described in the previous section, most buffer overflow exploits manipulate the return address on the stack to set it to a desirable value. Return addresses are a fundamental property of how computer programs operate with subroutines and cannot realistically be removed. The first logical step in countering buffer overflows is to ensure that return addresses only point to trusted program code and not to hostile externally injected program code. This is the approach that ExecShield, NX technology provided by AMD and Intel, and Microsoft's Data Execution PreventionTM all take.

The casual observer might wonder why these specific technologies are needed to create this behavior and why they are not there from the start. The reasons go to the roots of the 80x86 architecture. In the 80x86 architecture, there is no distinction between having permission to read from or execute program code from a certain part of memory. While this was done for full compatibility with the Intel 8086 processor from the 1980s, clearly this behavior is suboptimal.

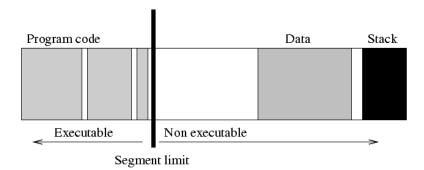
The Segment Limit Approach

ExecShield (and similar technologies such as PaX^5 approximate a separation of read and execute permissions by *segment limits* (an obscure feature from the Intel 80386 line of processors). The effect of applying segment limits is that the first N megabytes of the virtual memory of a process are executable while the remaining virtual memory is not. The operating system kernel selects the value of N.

⁵ The PaX homepage is at http://pax.grsecurity.net/







With such a segment limit in place, the operating system must make sure that all program code is located below this limit (seen at left in the Virtual Address space image) while data, especially the stack, should be located in the higher virtual memory addresses (the right side of the picture). When a violation of the execution permission happens, the program triggers a segmentation fault and terminates, this behavior is identical to when a program tries to violate read or write memory permissions.

Segment limits achieve a reasonable approximation of separated read and execute permissions with this divide-in-two approach. For virtually all applications, the kernel places the program code in the lower part of virtual memory while keeping the data mostly separate from this part. In some rare and special circumstances (for example with the XFree86™ server which takes care of the graphics in Linux) there may be less than optimal protection due to this approximation.

Intel and AMD NX Technology

Both AMD and Intel have recognized the lack of ability in separating read and execute permissions in the x86 architecture. In the AMD64 processor line, AMD extended the architecture in a backward compatible way by adding a *No eXecute* permission to the set of existing memory permissions. As with the existing read and write memory permissions, the kernel can control ${\tt No}$ execute permissions of the programs' virtual memory with a 4 KB granularity. Intel and other x86 processor manufacturers have announced their support for this NX technology in future product releases as well.

Using NX technology is a more fine-grained approach than the previously mentioned segment limits approximation. Therefore, ExecShield in the Red Hat Enterprise Linux v.3, update 3 kernels will use NX technology instead of segment limits when available in the hardware. In addition, NX technology also works for kernel mode unlike the segment limit approach.

There is one caveat concerning NX technology. As more permission bookkeeping information is required, this only works in the PAE 64 bit pagetable format (the PAE technology allows 32-bit x86 processors to use more than 4 GB of physical memory). PAE pagetables are not supported by all

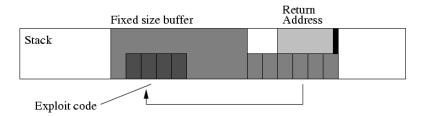


variations of x86 processors. They also incur a 6% overhead (approximately) on the systems performance. As a result of these restrictions, PAE and NX technology in Red Hat Enterprise Linux v.3 are only used in the kernel-smp and kernel-hugemem kernels (which already enable PAE for supporting more than 4 GB of memory), not in the uniprocessor kernel.

Randomization

Earlier in this document we described that a typical buffer overflow works by overwriting the return address on the stack to point into the buffer. For example:

Figure 4. Buffer overflow



To make this work, the attacker must know the approximate address of the buffer on the stack so that the return address can be filled with the correct address of the exploit code. While it may sound difficult to find this address, in practice it is quite easy to obtain. Each system running Red Hat Enterprise Linux v.3 basically has the same applications, binaries, and libraries. As a result of these similarities, the sought-after address is very similar or identical for many of the Red Hat Enterprise Linux v.3 systems. A person who is writing an exploit only has to examine his own system to determine the address that will be similar on all other such systems. This is not unique to Red Hat Enterprise Linux v.3--every operating system with a significant installed base will have this "same environment" problem.

Another approach in exploiting buffer overflows also involves overwriting the return address⁶. However, rather than overwriting it with the address of code in the buffer, it overwrites it with the address of a subroutine that is already present in the application, quite often the <code>system()</code> function from the glibc library. Since this type of attack does not depend on executing code in a data/stack area but does depend on executing previously present and legitimate code, it defeats the ExecShield approach of making the stack non-executable. Note, however, that this approach also depends on knowing the exact address of the function that is to be called.

Based on the observation that both ways of exploiting buffer overflows requires knowledge of exact memory addresses, the ExecShield technology in Red Hat Enterprise Linux v.3, update 3 gives randomized offsets to the addresses of

⁶ This class of exploitation technique is described in more detail at http://www.phrack.org/show.php?p=58&;a=4



several key components. This randomization offers more system security by making it nearly impossible to find the exact address needed for these exploits; the address is now different for every machine as well as being different each time a program starts.

In Red Hat Enterprise Linux v.3, update 3, ExecShield randomizes the following components of a program automatically:

- · The stack itself
- · Locations of shared libraries
- Start of the programs heap

Remaining Randomization: PIE Binaries

Until now, we have not discussed the randomization of the address in memory of the application code itself. In Linux, the application is traditionally compiled in such a way as to only work at a preselected address that has been selected when the binary was created. As such, this address cannot be reasonably randomized by the kernel. This randomization omission is a weak spot in the protection against buffer overflows.

Red Hat has developed, as well as contributed to the GNU Compiler Collection toolchain, a technique called Position Independent Executable, or PIE. This PIE technology is designed to overcome this limitation. PIE binaries are compiled in a special way, in that they are freely locatable throughout the entire address space of the program. In kernels with ExecShield technology, the addresses at which these binaries are loaded is randomized, yet in kernels without ExecShield and PIE support, these binaries are loaded at a generally predictable location. Not all programs are suitable to be compiled as PIE; because PIE binaries must be relocatable, they are compiled to become Position Independent Code (PIC). PIC has a small but measurable runtime overhead which is the same as the overhead of a shared library (since shared libraries are also compiled as PIC).

To compile a binary as PIE, the command line arguments -fpie and -pie must be passed to the compiler and linker respectively.

Since PIE is a new technology, not all programs in Red Hat Enterprise Linux v.3 use it yet. Red Hat expects to increase the usage of PIE in future updates of Red Hat Enterprise Linux v.3. In update 3, several highly visible programs are already compiled as PIE, including:

- The Bind DNS nameserver
- The Samba SMB/CIFS fileserver
- The Squid HTTP proxy
- The vsftpd FTP server

Additionally, several lower profile programs have been compiled as PIE (rusers and cron, the NFS RPC daemons).



To check if a binary is compiled as PIE, use this command:

```
readelf -h -d /usr/sbin/smbd | grep 'Type:.*DYN'
Type: DYN (Shared object file)
```

Compatibility

During the design and implementation of the new security features in Red Hat Enterprise Linux v.3, update 3, compatibility with existing setups and applications was a high priority in making the changes non-disruptive.

The main compatibility issue with making the stack non-executable centers around applications that need the stack or heap to be executable. This behavior may be needed for a variety of reasons, including cases of incorrect assumptions in the application (certain JVM's suffer this), or in cases where they are written using certain programming constructs or programming languages that in practice require the stack to be executable.

The solution to this compatibility problem lies in a special flag inside program binaries and libraries. This flag indicates whether or not the program or library in question requires the stack to be executable due to the usage of certain programming language features or programming languages. Red Hat contributed the code to make the the GNU Compiler Collection toolchain emit this flag correctly automatically when such programming constructs are used. In addition, if a binary or library does NOT have this flag, the kernel assumes this to be a legacy binary with unknown requirements with respect to stack executability, and will thus allow the stack to be executable to achieve maximum compatibility.

Table 1. Overview of ELF flag behavior

Flag value	Resulting behavior
Flag present, RWE	Stack is executable
Flag present, RW	Stack is not executable
Flag absent	Stack is executable

This scheme provides maximum compatibility for existing applications while providing maximum security for new known and valid applications.

For more technical details on how to create binaries with the correct setting of the flag see http://people.redhat.com/drepper/nonselsec.pdf, appendix A. This document also explains how existing binaries can be treated to get the protection.

To check if a binary or library has this flag with the eu-readelf program, use this command



which shows that the /bin/true binary in this system has the flag with value "RW". This binary runs with the stack set as non-executable.

An example of a binary where the source code uses programming language features requiring an executable stack is below:

If a binary or library shows no "GNU_STACK" line, it does not have the flag and runs with the stack executable.

System administrators who want to identify which processes on their live systems run with and without stack execute protection can use the Isexec shell script available online at http://people.redhat.com/arjanv/lsexec.

How Well Does It Work

An early version of the ExecShield technology appeared in the first release of the Fedora project. The second release, Fedora Core 2, has a more advanced implementation that matches the implementation now added to Red Hat Enterprise Linux v.3, update 3. The exploit prevention has shown to work well on microprocessors with and without NX technology.

In the period from November 1, 2003 to August 11, 2004, there were 16 security issues published in Linux that are more severe than a Denial of Service problem and for which an exploit was made available.

Table 2. CVE identifiers of security vulnerabilities with exploit

CAN-2004-0597	CAN-2004-0600	CAN-2004-0541	CAN-2004-0415
o = o o . o o o .	000 . 0000	07 1 = 00 . 00	07 1 = 00 . 0 0
CAN-2004-0397	CAN-2004-0424	CAN-2004-0409	CAN-2004-0396
CAN-2004-0176	CAN-2004-0084	CAN-2004-0077	CAN-2003-0985
CAN-2003-0963	CAN-2003-0962	CAN-2003-0961	CAN-2003-0328
CAIN-2003-0963	CAN-2003-0962	CAIN-2003-0961	CAIN-2003-0328

Out of these 16 exploits, four were for kernel security holes for which ExecShield does not offer protection, 11 were stack buffer overflows which are stopped by ExecShield technology, and one was a heap buffer overflow also stopped by ExecShield. Using ExecShield technology yielded a success rate of 75%. Note that these exploits were stopped even on microprocessors without NX technology.

It is important to note that ExecShield can only reduce the risk and impact of buffer overflow type security issues. The presence of these technologies should never be seen as a substitute for applying security updates provided by the operating system vendors.

Future Work



Red Hat is planning to integrate *SELinux*, the security framework being productized in cooperation with the National Security Agency, into the next major release of Red Hat Enterprise Linux. SELinux is an advanced technology designed to contain security breaches by being the best of breed in role based access control, mandatory access control, and discretionary access control. By implementing SELinux, processes are isolated from one another. Each process is forcibly restricted to only perform the operations it is expected to perform by a system wide security policy. While ExecShield is designed to prevent certain types of security breaches from happening, SELinux is an invaluable technology for the containment of security breaches in general; such layering of security technologies is called defense in depth.