Memory Protection with PaX and the Stack Smashing Protector

Breaking out Peace

The PaX kernel patch helps to bring a little peace of mind to safety-onscious users. Adamantix developer Peter Busser explains the basic principles and justifies the inclusion of PaX as an Adamantix component .

BY PETER BUSSER

Much has been said and written about securing systems by limiting what processes can do to other objects on the system, such as files, devices, shared memory, etc. This is designed to improve the integrity of the system and the confidentiality of data, and enhance overall availability of services on the system. Ample evidence for this concern can found in the number of Linux kernel patches and programs that exist today. One example is RSBAC [3], which is used by Adamantix [6]to achieve these aims.

Much less has been written about the protection of the only active objects on Linux, that is, the internal protection of processes. Processes fulfill an essential role on Linux, because they are the only place on a Linux system where code can be executed. Despite this importance, the Linux world has a tradition of ignoring process memory protection. This has resulted in a large number of exploits that depend on process memory corruption.

Why is memory protection important?

In a perfect world, software would have no bugs and therefore attackers would not be able to get read/write access to internal process memory. There are people who claim that we could enter this perfect world if we only stopped developing programs in C and instead used Java, C-LISP, Ada, or whatever other language they think is best. If life were only that simple!

Even if this is true, and this still remains to be seen (don't forget that Java JDK, Perl, Python, etc. all depending on interpreters, which are written in C and often introduce their own security problems), and we started to convert C source code to the ultimate safe language XYZ, it would still take many years to finish this work. In the meantime, systems would still depend on vulnerable C code.

We have to accept the fact that the world is not perfect, and that even the best programmers make mistakes. As a consequence, the internal process memory can be exposed to corruption by outside attacks, and therefore needs protection.

History

There are several reasons for the lack of attention to process protection and the general failure to understand its significance. Traditional security research has tended to focus on access-control and encryption mechanisms. Process integrity did not really start to become an issue until buffer overflows were discovered and became increasingly popular over the years.

Linus Torvald's dismissal of the OpenWall non-executable stack patch [1] was an important event in the history of Linux. The OpenWall patch implemented a non-executable stack for the Linux kernel. But the stack is only one of several process memory areas that need protection. Linus's stance, that protecting only the stack does not provide much security, was justifiable. Unfortunately, this made many people believe that ALL process protection mechanisms are useless. This is as wrong as believing that the Earth is flat, but nevertheless it is still a widespread, and extremely persistent belief.

In the days when computers were still personal computers, security simply did not matter. As long as you did not connect a computer to a network, and could keep other people away from the machine, it was pretty much safe. Many of today's programmers grew up in such a world. With the growth of the Internet, the situation has changed dramatically. So far it has been easier to ignore the problem and tell people that it is no big deal. But the problem can no longer be ignored.

PaX enters the arena

There have been several efforts to create improved memory protection patches since the OpenWall patch was dismissed by Linus. The PaX homepage [2] has links to some of them. Development on most of these patches has been abandoned, basically leaving PaX as the only one under active development. Thanks to the persistence of the author of PaX, Linux users can now enjoy the best memory protection available in the free software world.

Work on PaX started some 30 months ago. After examining many exploits, the PaX author came to the conclusion that improving memory protection was the only way to stop many of them. But PaX did not see widespread use, until the gr-security patch started to include PaX. Gr-security is well known and many people use it.

Several distributions, like Gentoo hardened, support gr-security and therefore PaX. This was a boost to development; people started to request features, reporting bugs, and helping to port PaX to other architectures. There is now a small but active community of people who use and support PaX.

The difference between PaX and other memory protection patches is that PaX does not try to prevent specific exploits. It tries to prevent certain classes of exploits. Animals and plants are catagorized by reference to their similarities. There are several classes of animals: mammals, birds, fish, etc.

The same can be done with exploits and attacks. Now suppose someone invents a fence that protects your land against wild boars. Really useful if you have a crop that is frequently ruined by these animals, but the same fence may not be strong enough to keep out elephants, or the holes may be too big to keep out rodents. A fence which provides protection against ALL walking animals, would clearly be a superior fence.

In the free software world, PaX is like that fence. It provides protection against a whole class of exploits. Or, in other words, it provides protection against ALL walking animals. But not against birds or insects. Other memory protection solutions may only protect you against certain animals. You could say that OpenWall only protects you against rodents. And OpenBSD's W^X [4], and exec-shield [5], protect you against all walking animals, except elephants. But more of that later.

Different classes of attack

Generally speaking, there are three classes of attack that memory protection patches try to address:

(1) Introduce and execute arbitrary code.
(2) Execute existing code out of original program order.
(3) Execute existing code in original program order with arbitrary data.

Every possible memory corruption exploit belongs to one of these three classes. For example, many popular buffer overflow exploits belong to (1). The example quoted by Linus Torvalds, used the so-called return-to-libc style technique. It belongs to (2). Class (3) exploits exist, but are rare. It is normally easier to use class (1) or (2) instead. Note that this is a classification of exploit techniques, and not bugs. That is, a given technique can be used to exploit different bugs, and a given bug can be exploited by more than one technique.

The idea behind PaX is to prevent entire classes of exploit techniques from working. At this moment, (1) has been dealt with (i.e. this is the fence against all walking animals), (2) is in the pipeline, and (3) is somewhere in the future (pending some research). What sets PaX apart from other memory protection patches is the fact that they do not deal with (1) completely, and often ignore (2) and (3).

Class (1)

Introduction and execution of arbitrary code means that it is possible to:

- Overwrite code that is loaded in memory,
- Overwrite data that is in memory, and execute it as if it is code,
- Load new code from disk in memory and execute it.

If it is possible to overwrite code, an attacker can inject her own code into the execution flow of a process. Instead of doing what the program was designed to do, the process starts to do what the attacker wants.

The same goes for the second technique, where data is overwritten and then executed as if it were code. Although you might assume that a system would handle

grsecurity - Mozilla1
File Edit View Go Bookmarks Tools Window Help
http://www.grsecurity.net/
Search
grsecurity
www.grsecurity.net
news papers download features links mirrors
mailinglist forum cvs donations lsm contact

For the latest stable or development release, goto the download section.

Latest stable release is: 1.9.13 .
Latest development release is: 2.0-rc4 .

Figure 1: Many developers have started taking the Pax memory protection system more seriously, now that it has been incorporated in GR-Security.

code and data in separate ways, programmers prefer the convenience of treating code and data identically. Of course, attackers find this just as convenient. As a result, this technique is used by most buffer overflow attacks. PaX ensures that code is code and data is data. So you can read and write data, but not execute it, and execute code, but not write to it.

The first two techniques only require write and execute access to memory. PaX can deal with these techniques on its own. The third is different in that it also requires file access. Electricity is often used to make fences more effective. PaX does something similar. By leveraging ACLs and/or other access control mechanisms, like for instance RSBAC [3], PaX can guarantee perfect protection against all class (1) attacks. It is the only memory protection patch that comes with a guarantee of this kind.

Class (2)

Class (2) includes the example used by Linus Torvalds to dismiss the OpenWall patch. In general it means that the attacker overwrites an address in memory that is used to control how the process works. For instance, overwriting the return address on the stack, so that when a function returns, it does not return to the place where it was called, but to a place the attacker has chosen.

But there are several other places, used for various purposes, where addresses are stored. And in theory, all of these places can be used by attackers to influence a process.

Class (3)

This class includes attacks which overwrite important data. In cartoons there is often a scene where the hero puts on a disguise and sends the bad guy off in the wrong direction. Similar tricks can sometimes be used to attack programs. If an attacker can somehow overwrite important data, the program can be tricked into going down the wrong logical path and doing unexpected things.

Take the mount command, for example. Mount can be configured to allow certain users to mount disks. It performs some checks to discriminate between users who are allowed, or not allowed, to mount.

If the attacker can somehow influence these checks, then the mount command will believe that the attacker is okay, and it therefore mount a disk (which then can then be used to launch the next attack). This is a purely hypothetical example. Good examples are hard to find, because this class is harder to find and exploit.

Combined protection mechanisms

Protection against only one class is useful but not very effective. Linus used a Class (2) example to circumvent the Class (1) protection of the OpenWall patch. This is true for all classes; you can avoid protection against attacks in one class by using a technique from a different class.

People often look at only one class of attack, and forget the others (like the OpenWall patch does). And what's worse, people dismiss the usefulness of protection against one class, because additional protection mechanisms are needed to prevent attacks in the other classes. This is the mistake Linus Torvalds made. Combining protection mechanisms against all three classes of attacks is the only correct answer here.

Stack Smashing Protector

The combination of defense mechanisms against the different classes of attack is one of the reasons that the Stack Smashing Protector (SSP) was added to Adamantix. SSP [7] (also known as ProPolice) is a GCC patch which does several things to prevent a limited number of Class (1), (2), and (3) attacks known as stack overflows.

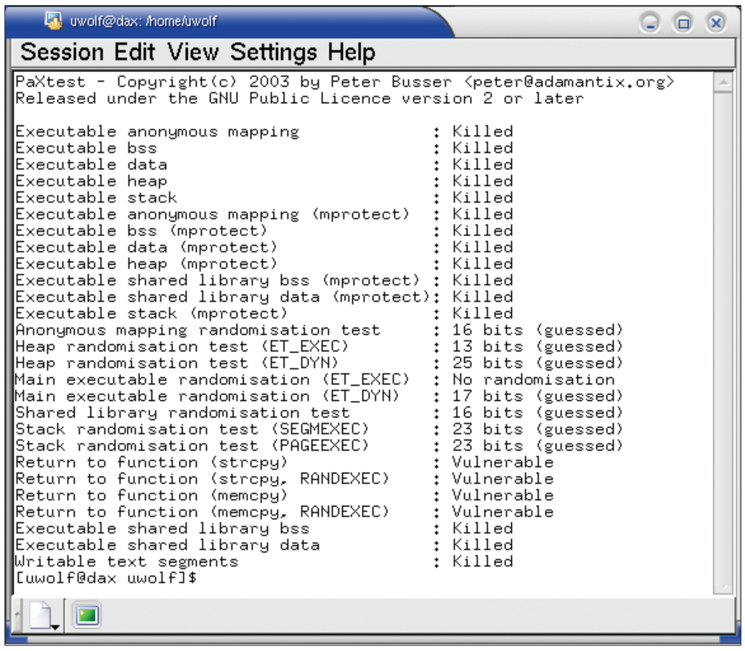
SSP uses two mechanisms to accomplish this:

- It adds a booby trap to the stack when it detects a function that is potentially dangerous (however, the detection mechanism is not foolproof).
- It changes the order of local variables, so that dangerous variables are placed next to the booby trap, thereby increasing the chance of detection.

The booby trap is often referred to as a "canary", after the canary birds that were used by coal miners to detect lethal CO gas. The gas would kill the bird before it could kill the mine workers. The canary in this case is a random number that is placed on the stack. A stack overflow attack will overwrite the number, which is then detected by SSP before the exploit code is executed. SSP writes a message to the system log and terminates the program if it detects that the random number has been overwritten.

This kind of check is quite expensive, especially when small functions are used. SSP tries to detect functions that could be vulnerable to stack overflows and only adds the checks to these functions. The detection mechanism is not perfect, so it is likely that SSP will fail to add checks to functions that need them, while adding them in places where they they are not needed.

SSP can also be used to compile the kernel, adding



```
PaXtest - Copyright(c) 2003 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable anonymous mapping (mprotect)  : Killed
Executable bss (mprotect)                : Killed
Executable data (mprotect)               : Killed
Executable heap (mprotect)               : Killed
Executable shared library bss (mprotect) : Killed
Executable shared library data (mprotect): Killed
Executable stack (mprotect)              : Killed
Anonymous mapping randomisation test     : 16 bits (guessed)
Heap randomisation test (ET_EXEC)        : 13 bits (guessed)
Heap randomisation test (ET_DYN)         : 25 bits (guessed)
Main executable randomisation (ET_EXEC)  : No randomisation
Main executable randomisation (ET_DYN)   : 17 bits (guessed)
Shared library randomisation test        : 16 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 23 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 23 bits (guessed)
Return to function (strcpy)              : Vulnerable
Return to function (strcpy, RANDEXEC)    : Vulnerable
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, RANDEXEC)    : Vulnerable
Executable shared library bss            : Killed
Executable shared library data           : Killed
Writable text segments                   : Killed
[uwolf@dax uwolf]$
```

Figure 2: The output from Paxtest shows that the Pax-hardened kernel is immune to most attacks.

stack overflow protection to the kernel. And thanks to the optimizations performed by SSP, performance is still good.

More elaborate compiler techniques exist to make C programs more secure, such as full bounds checking. Bounds checking means that all access to data is checked. This provides much more security but also a lot of overhead. The speed would be comparable to that of Java. This is really a big performance hit that few people are willing to take.

Randomisation

A feature found in PaX and other memory protection patches is Address Space Layout Randomization, or ASLR. ASLR means that different parts of a program are loaded at different places in memory. The memory layout changes each time a program is executed.

This is not a protection mechanism. It does not provide any kind of control mechanism. However, it can make succesful exploitation more difficult if the attacker can not work out the exact memory layout. The various memory protection patches differ in the amount of randomization they provide. More randomization is preferable, because it makes brute force attacks harder. Currently, PaX provides the best randomization of all the available memory protection patches for Linux and also better randomisation than OpenBSD.

Compatibility

It is not necessary to recompile programs to use them on a PaX kernel. Most programs will run fine without any modifications. Problems are mostly caused by libraries. From experience I know that Debian Woody has some issues with libraries; only a few libraries, but important ones (such as zlib). Other people reported that older Red Hat versions had many library-related problems (I have had no reports about newer versions of Red Hat). I have briefly tested Debian Sarge and, except for XFree86, it worked fine on a PaX kernel.

Most programs work fine on PaX, even with the most restrictive settings enabled (like the ones used by the Adamantix kernel packages). Of the several hundred packages that have so far been adapted for use on Adamantix, only a handful cause problems with PaX. Most of these cases were trivial to fix. The most frequent changes required are compilation flags, especially for libraries. Sometimes this meant using C, instead of assembly code (e.g. for zlib and gnupg). And sometimes a few lines of the source code had to be changed (because of how the C compiler works).

Only a few programs cannot be fixed. That is because they depend on creating executable code in memory. The SUN Java Runtime Environment (JRE) is a good example of this.

You can use chpax to define exceptions for this kind of program. The chpax settings are saved inside the executable. When the executable is started, PaX will detect these settings and disable some of the checks. Most problematic programs can be made to run this way without having to change the code.

Paxtest

When PaX was added to the Adamantix kernel and I started to recompile/relink programs for PaX ASLR, it worked so beautifully that I started to doubt if PaX did anything useful. Before I started, I had expected many programs to break, and instead nothing serious seemed to be happening. How was that possible? One possibility was that PaX did not work. Instead of speculating, I decided to find some proof, and this is why I started to write paxtest.

Paxtest is a collection of test programs that each check one functional aspect of PaX. One test writes machine code to a character string and then tries to execute that code. A properly working PaX configuration will detect this and kill the program. Other tests check other protection features.

There are also tests that measure ASLR randomization. Together they provide some information about the level of protection. The more tests that say “killed”, the better. Using the output from paxtest, I found out that PaX was getting along just fine with the Adamantix kernel, and that the lack of problems was a sign that PaX was working much better than I had expected.

Comparison between PaX and other patches

The nice thing about paxtest is that it can also be used to test the memory protection provided by other patches. Anyone can download paxtest from the PaX web site [2], and then compile and run it, to compare results. (Or apt-get install paxtest if you are using Adamantix). Running paxtest on a kernel patched with OpenWall shows only a non-executable stack, which is to be expected. In other words, it provides hardly any protection.

Running paxtest on exec-shield returns different results, depending on the version of paxtest you use. Bugs in older versions of paxtest made people believe that exec-shield performed better than it actually does.

In general, the protection provided by exec-shield is rather “underwhelming”. And it will be interesting to watch how fast the exploit writers will be able to adapt to the weaknesses in exec-shield. It would be nice if someone could port paxtest to OpenBSD. I do not think that OpenBSD’s W^X will return really convincing results.

Conclusion

Process memory protection is important, but so far there has not been a solution that provides protection against all three classes of attack. The oldest memory protection kernel patch, which also provides the best protection at this time of writing, is PaX. With the help of ACLs or other access control mechanisms, it can guarantee perfect protection against Class (1) attacks.

Other mechanisms, such as SSP, are needed to defend against Class (2) and (3) attacks. The cost of implementing this is relatively low. Any Linux distribution that cares about security should plan to include this kind of protection. ■

INFO

[1] *http://old.lwn.net/1998/0806/a/linus-noexec.html*

[2] *http://pageexec.virtualave.net/*

[3] *http://www.rsbac.org/*

[4] *http://archives.neohapsis.com/archives/openbsd/2003-04/1362.html*

[5] *http://people.redhat.com/mingo/exec-shield/*

[6] *http://www.adamantix.org/*

[7] *http://www.research.ibm.com/trl/projects/security/ssp/*