Why hardening your system is a good idea.

Francisco Blas Izquierdo Riera (klondike(a t)xiscosoft.es)

Why protecting your binaries is a good idea?

Most statistical studies on common vulnerabilities on binaries[1] show that buffer overflows[2] are both, the most frequent software flaw and the most exploited ones. The effects of these kind of flaws vary depending on the defense measures enabled and the place where the buffer is (stack space, mmaped space, global space or heap space). In any case the consequences range from simple Denials Of Services to the Execution of the desired code with the same privileges as the user running the program (or kernel level[3] if the bug is in the kernel space). As examples of infamous programs designed to exploit those kind of flaws we can talk of the Blaster and Sasser worms, the exploits on the GDI+ library or any exploit for the 2008 ndiswrapper bug [4].

On the other hand, another common mistake is improperly dereferencing user space pointers (like NULL), on kernel space, which could be easily exploited by a local attacker to execute code with kernel level privileges. A clear example of this kind of bug is the one discovered on some of the network protocol modules on the Linux Kernel[5] or the recent ia32 syscall exploit[6].

How do these exploits work?

In order to know how the defenses act we should first know how these two common kinds of attack vectors work. In the following lines we'll explain how them work on the common x86 and amd64 architectures, though these kind of failures also affect other architectures.

Buffer overflows

The inner of these kind of attacks vary according to the place where the buffer is placed (stack space, mmaped space, global space or heap space), anyways in all the cases the exploitation consists on inserting a string longer than the allocated space as input so a part of it is written outside of that reserved space overwriting other data structures with the data we want. On the next chapters we'll explain how these exploits work on different cases so we can understand how the protection mechanisms work later.

1 One example of then is at http://www.qualys.com/research/rnd/top10/

2 It must be noted that this is only the case for binary applications, for example an apache web server. On web applications, the most common security flaws are usually of other kind as can be checked at http://www.owasp.org/index.php/Top_10_2010-Main for example. In that case, the most frequent vulnerabilities are code injections and XSS (cross-site scripting).

3 Kernel level privileges implies complete access to hardware and the whole memory space so a bad intended user could even do physical hardware to the machine, for example, reflashing the BIOS as some old viruses did.

4 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4395

5 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692

6 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3301

Buffer on an executable stack

Although this kind of case is, nowadays, one of the most uncommon, as stacks are now usually marked as non executable, it has a space here because the methods explained here were the most common until a few years ago (as a matter of fact this kind of attack is still very frequent on some beginner's security challenges competitions), and to some extent they can still be exploited on some machines (for example on some binaries with trampolines).

The usual proceeding in this case consists on writing some binary code to execute a exec syscall using as arguments the ones we want and preceded by a lot of 1 byte nops so we have some margin. Also the function return address is overwritten so it points somewhere on the middle of our nops.

To see how it works give a look to the attached table.

Word Meaning	Stack before the attack	Stack after the attack
Parent data	Unknown	Unknown
Parent data	Unknown	Unknown
Argument2	2	2
Argument1	1	1
Stored Frame Pointer	Parent FP	Parent FP
Return Address	Parent Return Address	&buffer
Local var3	1	shellcode
Local var2	2	shellcode
Local var1	3	nop, nop, nop, nop
Buffer3	Unknown	nop, nop, nop, nop
Buffer2	Unknown	nop, nop, nop, nop
Buffer1	Unknown	nop, nop, nop, nop

As you can see when the function returns it will continue executing code around the place where buffer1 is so it will execute some nops and then our shellcode[7] which could for example launch a bash shell.

7 A shellcode is a small piece of binary code used to make the program execute another binary image with the desired arguments.

Buffer on a non executable stack

Here we can try two different attacks, the first is known as ret2libc and consists on overwriting the return address with that of a function on a loaded library and the arguments with pointers to the strings we want to execute. The following table shows how such an attack would work:

Word Meaning	Stack before the attack	Stack after the attack
Parent data	Unknown	Unknown
Parent data	Unknown	Unknown
Argument2	2	2
Argument1	1	&buffer1
Stored Frame Pointer	Parent FP	a,a,a,a
	Parent Return Address	system
Local var3	1	a,a,a,a
Local var2	2	a,a,a,a
Local var1	3	a,a,a,a
Buffer3	Unknown	h,\0,a,a,
Buffer2	Unknown	/bas
Buffer1	Unknown	/bin

As you can see when the function returns it will call system using /bin/bash as argument. Although this is a very basic approach, there are more complex and better shell code generators for these kind of attacks (for putting the string before the arguments in the stack so they won't be overwritten).

Also another kind of attack will be overwriting a local variable to control the program flow.
Let's suppose that Local var1 is a boolean flag telling whether the authentication has succeeded on a pam module. We can see the attack on the following table:

Word Meaning	Stack before the attack	Stack after the attack
Parent data	Unknown	Unknown
Parent data	Unknown	Unknown
Argument2	2	2
Argument1	1	1
Stored Frame Pointer	Parent FP	Parent FP
Return Address	Parent Return Address	Parent Return Address
Local var3	1	1
Local var2	2	2
Local var1	0	1
Buffer3	Unknown	a,a,a,a
Buffer2	Unknown	a,a,a,a
Buffer1	Unknown	a,a,a,a

As can be seen after the attack the module will suppose the user has authenticated correctly because the flag is set to 1.

Buffer on mmaped space

The attack will vary greatly depending on the flags for the mmaped space and the position where it is loaded. If it is near the got table, for example, we could overwrite it making the global variables point to wherever we want changing the execution flow of the program. If it is near of the heap space we can proceed on a similar way to that used for heap variables, if it is near of the stack we can proceed as if it was a stack buffer overflow, etc...

The most dangerous case comes when the mmaped memory is executable and is going to be executed by the program because in that case we can simply load our shellcode there.

Buffer on global space

In that case as globals are before heap space, we can either corrupt other global variables changing the execution flow of the program or even corrupt the heap data structures and do an attack similar t those based on heap space

Buffer on heap space

This are usually the most hard to exploit due to the properties of the double linked list used on the heap. In any case, the idea here consists on overwriting some of the heap control data structures in order to make them, for example, deallocate portions of memory which shouldn't be deallocated amongst other things.

User space pointer dereference

This kind of attack consist on allocating some space on the dereferenced address (0 in the case of NULL) either executable or simply READ/WRITE and load the appropriate data in that space. Depending on the vulnerability, the address will be either a constant address (like NULL) or an address set by the program when executing the exploit.

User space data pointer dereference

Though this is a fairly uncommon kind of attack, it simply works by loading the desired data on the pointed address so the kernel reads it altering the kernel program flow from user space.

User space function pointer dereference

In this case the program will either load a file with the code to be executed or write it into the address. This kind of attack works for NULL pointers because on some structures with function calls those functions point to NULL when the function is not set and the kernel programmers expect the calls to fail when called because of this. But, if a page has been loaded at address 0 then the call will succeed and the execution flow will continue there with the code loaded by the attacker.

Other way in which this works is if the input data is not properly validated on entry points resulting in jumps to the address indicated by the bad formed request.

Which is the risk of each kind of attack?

To show the damage which could be done by each kind of attack the following table is added. There the risk is evaluated from 0 to 10 according to how easy exploitation is and how probable is that the bug could result on that kind of exploitation is. In some case a range of values is used as the values vary a lot.

Exploit	Arbitrary Code Execution	Out Of Order Code Execution	Program Flow Alteration	Denial Of Service[8]
Executable stack overflow	10	10	10	10
Non-executable stack overflow	7	8	10	10
Mmaped space overflow	0-10	0-3	7	10
Global space overflow	3	4	10	10
Heap space overflow	3	4	10	10
User space data pointer dereference	3	4	10	7
User space function pointer dereference	10	10	10	10

As we can see the risks, are usually very high. So protecting against them is a good idea. On the next chapters we'll explain why Gentoo Hardened is a good choice to have some protection against them.

Why Gentoo Hardened?

Gentoo Hardened has, for years, been one of the pioneers on the integration of bleeding edge protections against those, and others, known common vulnerabilities. For that, Gentoo Hardened uses a modified gcc toolchain which hardens binaries at compile time and a set of kernel patches, known as PAX which add various levels of protection against those common attack vectors.

Also, following Gentoo's philosophy of letting the user choose which things he wants to enable, it allows the system administrator choose which of these hardening features will be present and which won't, being able even to enable or disable the PAX related ones on a per binary basis (using paxctl or RBAC).

Finally, the Gentoo Hardened team is eager to help anybody with troubles as long as they have taken sometime to read the documentation provided by the project.

Summing it up, Gentoo Hardened is a good choice because of its long experience, because of their efforts to keep introducing bleeding edge technologies to make exploiting more difficult (whilst allowing you to choose which ones you want to use) and because its experienced team will be glad to give you a hand if you have troubles.

8 Please not that in this case we talk of Denial Of Service against the service provided by the affected binary, and not against the whole system (except on the user space pointer dereference cases).

What are the hardening features it provides?

As said before, Gentoo Hardened is a project known for their efforts on integrating new hardening technologies. As such, the project was the first offering SSP support (which is now provided by some of the popular distributions) and PIE support for ASLR (which nowadays is used even by Microsoft Windows). Also, using PAX they provided support for NX (now also provided by Microsoft Windows but only on some processors).

On the next chapters we'll explain how they work protecting against the common exploits commented before.

Compilation time protections

A well layered protection system must be able to act against known and unknown attacks on as many levels as possible. In this case, some of the protections can only be enabled at compilation time as they depend on some knowledge of the source code, add some code or modify the way the code is built to satisfy some requirements.

SSP

SSP works by adding a random canary on the stack before the local variables, either only on functions deemed dangerous or on every function (which is the default option on Gentoo Hardened), and checking its value before returning from the function. To do so it tries to add the minimal amount of code possible to avoid new flaws and to require as few processor time as possible.

Here we can see one of the previous examples before and after the overflow happens:

Word Meaning	Stack before the attack	Stack after the attack
Parent data	Unknown	Unknown
Parent data	Unknown	Unknown
Argument2	2	2
Argument1	1	&buffer1
Stored Frame Pointer	Parent FP	a,a,a,a
Return Address	Parent Return Address	system
Canary	Random Value	a,a,a,a
Local var3	1	a,a,a,a
Local var2	2	a,a,a,a
Local var1	0	a,a,a,a
Buffer3	Unknown	h,\0,a,a,
Buffer2	Unknown	/bas
Buffer1	Unknown	/bin

When the canary is checked the SSP code will find that the value isn't the same and will kill the program reducing an arbitrary code execution to a DOS.

-D_FORTIFY_SOURCE

This gcc compiler function works doing mainly three tasks:

1. **Dangerous variable reordering**: This results into a code where dangerous variables, like char buffers are put before other local variables making it impossible to overwrite them if an overflow happens.
2. **Compilation and run time overflow detection**: This results in some overflows, like writing 20 characters to a variable which statically allocated 16, being detected during compilation and run time so they can be easily fixed (or killing the process to avoid further damage).
3. **Warn on the usage of dangerous functions**: As a result when using functions known to be dangerous like, for example gets, the compiler will issue a warning to the user.

To have a clearer view of how reordering works we'll show how the local variable overwriting example seen before won't work at all on a setup with -D_FORTIFY_SOURCE and SSP enabled:

Word Meaning	Stack before the attack	Stack after the attack
Parent data	Unknown	Unknown
Parent data	Unknown	Unknown
Argument2	2	2
Argument1	1	1
Stored Frame Pointer	Parent FP	Parent FP
Return Address	Parent Return Address	Parent Return Address
Canary	Random Value	1
Buffer3	Unknown	a,a,a,a
Buffer2	Unknown	a,a,a,a
Buffer1	Unknown	a,a,a,a
Local var3	1	1
Local var2	2	2
Local var1	0	0

As we can see, not only is the local variable not overwritten, but also, the canary is modified so the application will be killed before the function returns.

PIE

PIE creates binaries which don't expect to find their symbols on specific positions. As a result, ASLR can be used with them resulting in the stack, the data, the heap, and the code segments being put on addresses chosen randomly (between some limits) making attacks depending on the knowledge of those directions much harder to succeed. This is specially effective on 64 bit architectures due to their amount of addressing space available.

PIC

PIC is the PIE counterpart for libraries. For ASLR to work smoothly the PIE binaries must be linked against PIC libraries (otherwise the linker will refuse to do the job). This works like PIE allowing ASLR to put the different segments of the libraries at random addresses.

Linking time protections

Most programs need to access to static data on libraries, or that a special table known as Global Offset Table (or GOT) is used. Also, the Procedure Linkage Table (or PLT) contains pointers to the various library functions used by the executable. If the attacker could overwrite any of them some of our defenses would be futile.

To avoid attacks targeted at these, and other, linking related structures, we have to harden also the linking stage.

RELRO

RELRO marks as read only all those data structures which won't be modified by the linker after the program starts running. As a result, the data on those structures won't be overwritable thus making the modification of their data impossible.

BIND_NOW

BIND_NOW instructs the linker to load all the symbols[9] needed by the application before it runs. As a result RELRO will be more effective protecting the whole GOT and PLT tables along with other data structures.

Kernel level protections

As stated before, for some of these protections to work some extra kernel support is required. Also, at this level some additional protections can be enabled to avoid some attack factors (for example we can mark the stack space as non-executable to reduce the stack overflow risks). Here we'll make a brief study of these defenses which are available thanks to PAX.

ASLR

ASLR (or Adress Space Layout Randomization) places the application sections randomly making it harder to guess their addresses and making attacks more unlikely to succeed. A clear example of an attack which is more unlikely to be successful is the ret2libc as the function address will be hard to tell because the code section will be allocated on a random place. Also, as the stack address will be randomized attacks based on knowing stack addresses will tend to fail more frequently[10].

RANDMMAP

RANDMMAP works on a similar way to ASLR randomizing the base address of the calls to mmap which don't have a fixed address. As a result, attacks depending on this knowledge are more likely to fail.

RANDKSTACK

Acts like ASLR randomizing the kernel stack base. Nowadays it can only be used on x86.

---

9 In this context, symbols refer to external resources offered by libraries like functions or global vairables.

10 Please note that the probability of success depends on the level of randomization and on hiding important data, like the addresses from the attacker.

UDEREF

UDEREF tries to avoid involuntary kernel accesses to the user space area, this makes user space function pointer dereferences impossible on amd64 and x86 and makes user space data pointer dereferences impossible on x86[11].

MIN_MMAP_ADDR

MIN_MMAP_ADDR disallows mmaps below the specified address, thus making kernel NULL pointer dereferences impossible. The bad thing is that disabling this is required at times, so root and the applications with the RAW_IO capability can override it.

NX Memory

PAX has a segmentation based NX implementation for older processors and a paging based one for the newer ones. This results on a real separation of code and data pages (even at kernel level on some architectures), protecting code from writes and avoiding the execution of data.

Mprotect restrictions

This complements the previous addition making it impossible, amongst other things, for an application, mark pages once writable as executable and write on executable pages. This way, new code is more unlikely to enter on the applications.

Kernel counter overflow protection

PAX protects the kernel from reference counter overflow producing a non-freeable object in case any happens instead of freeing its space allowing some exploits.

Free memory sanitization

This will make the kernel 0-fill freed pages so sensitive information can't be leaked to other processes.

Bounds checking on moves between kernel and userland

This ensures that when copying data from userland to the kernel an overflow can't happen and also that when copying data from the kernel to userland, no kernel data is leaked.

11 This may be ported to amd64 if someone figures out how to do the trick only with paging as AMD64 has no segmentation support.

GRSEC additional protections

Also, when using the Grsecurity kernel patches, various additional protections along with the MAC[12] can be enabled. Those restrictions are targeted more on limiting what users and root can or can't do, making exploiting even more hard.

Deny writing to /dev/kmem, /dev/mem, and /dev/port

This avoids raw memory reading and writing closing 3 ways of kernel level code insertion. Anyway, some programs (like X11) depend on it, so they are allowed to write, but only, on specified areas like the video memory.

Restrict VM86 mode

This makes enabling the Virtual 8086 mode impossible for processes without the RAWIO capability protecting against flaws on that emulation. It can only be enabled on x86[13].

Disable privileged I/O

This will disable the calls to ioperm and iopl which can be used to modify the running kernel. The bad thing is that they are required for X11 so if you want to enable it and use X11 you'll have to use the RBAC system.

Remove addresses from /proc/<pid>/[smaps|maps|stat]

This will remove the addresses from /proc/ when ASLR is enabled, so attackers can use that information when doing attacks like those based on buffer overflows.

Deter exploit bruteforcing

This will add a 30 second delay before the parent can spawn a new process if any of its children is killed by PAX or crashes because of an illegal instruction (those are signals SIGKILL and SIGILL respectively) making exploits using bruteforce attacks against forking daemons (like apache) take a lot longer to succeed. If your system is safe enough you can also add SIGSEGV modifying a bit the patch (though it can result in a DOS attack to the affected service if it has memory bugs). Bear in mind that with this option enabled, vulnerable services on your system would be more prone to a DOS attack (as the attacker won't need to kill the application so frequently to ensure the aren't childs left to serve legit petitions). Another alternative is developing a policy so preforked apps are restarted after logging those signals various times.

Harden module auto-loading

This will avoid the loading of modules by unprivileged users so they can't load buggy modules into the kernel and exploit them. This is done for example when a syscall to a function supported by a module is done.

---

12 MAC stands for Mandatory Access Control. In the case of grsec this means RBAC (role based access control) which allows a better access control to system administrators and also disable some of the PAX and grsec features on a per binary basis to avoid incompatibilities.

13 Bear in mind that support for the Virtual 8086 mode has been dropped on the AMD64 architecture.

Hide kernel symbols

This will make getting information on the loaded modules and kernel symbols impossible for unprivileged users. For this to be effective, you must not use a distribution precompiled kernel and you must hide files like the kernel image, the kernel config or the System.map. This makes harder for an attacker calling code in kernel space as it would require to know its position first.

Hide kernel processes

This will hide the kernel threads to all the processes except those with the “view hidden processes” flag. This will make it harder for an attacker to get information about possibly vulnerable processes or modules on the system.

Maximum password tries to access RBAC

This will block users who fail various times the password from trying to authenticate for a specified time.

Restrict /proc to user only

This avoids that users fiddle data from other users processes through /proc. Enabling this feature will make it harder for attackers to escalate privileges as they won't be able to know which processes are being run by other users. Also ensures a better privacy for the users on the system as an attacker wouldn't be able to see sensitive information which may be passed through the program parameters (for example).

/proc additional restrictions

This adds additional restrictions to avoid users viewing the devices information and slabinfo as it could be used to find exploits.

Linking restrictions

This avoids /tmp like race conditions because users won't be able to follow symlinks owned by other users in +t directories unless the symlink owner is the directory owner. Also will avoid users hardlinking files they don't own.

FIFO restrictions

Similar to the above, this disallows users who try to write on FIFOs they don't own unless the directory owner and the FIFO owner are the same.

Run time read-only mount protection

This will do the following after a sysctl flag (romount_protect) is set:

- Disallow new writable mounts on the system.
- Make impossible making current read only mounts writable.
- Denying any writes on block devices.

Chroot jail restrictions

These disallow certain actions to chrooted processes. In particular:

- Mounts.
- Double chroots. (If they are done to places outside of the current chroot).
- Pivot_root (as they can be used to break out if the function is used incorrectly).
- Fchmod and chmod +s (as this can be used to break out of the chroot and by other programs to gain privileges outside of the chroot).
- Fchdir out of the chroot.
- Mknod calls (as this can expose devices)
- Shmat calls pointing out of the chroot (as this can be used to access the shared memory of outside processes).
- Connecting to AF_UNIX socket not belonging to a filesystem that were created outside of the chroot (as this can also be used to access sensitive data of outside processes).
- Accessing to outside processes via kill, fcntl (to send signals), ptrace, capget, getpgid, setpgid and getsid.
- Viewing processes outside of the chroot.
- Changing the priority to processes outside of the chroot.
- Raising the priority of processes inside the chroot.
- Writing to the sysctl system.

Also will force a chdir to / if desired and remove certain capabilities like:

- Module insertion
- Raw I/O (which includes allocation of memory to the lower 64K)
- System taks
- Admin tasks
- Rebooting the system.
- Modifying inmutable files (which is very useful when sharing static files amongst chroots).
- Modifying the IPC of another process.
- Changing the filesystem time.

Enforce RLIMIT_NPROC on execs

This makes the system check the process limit also when executing a new task and not only when forking from an existing one.

Dmesg(8) restriction

This makes impossible acceding to dmesg to normal users thus making them unable to read the last messages from the log buffer of the kernel avoiding their usage by attackers to find vulnerabilities.

Deter ptrace-based process snooping

This makes ptracing of processes which aren't children of the current process impossible for non root users, deterring some malicious programs like TTY scanners.

Trusted path execution (TPE)

This allows various levels of execute access protection, allowing, amongst other things, having some users which can only execute programs on root owned directories (which must be writable only by root).

On one side you can choose a more lax mode which will make all users unable to execute things which aren't either on root owned directories writable only by root or directories owned by the user and writable only by the owner.

On the other side, there are two modes of operation on one you set a group who will be affected by TPE on the other (when using the Invert GID option) you set a group of users who won't be affected by the restrictive mode (but they will still be affected by the “Partially restrict non-root users” more lax mode if it is set).

TCP/UDP blackhole and LAST_ACK DoS prevention

This avoids some DoS attack coming from the network and makes the port scanners' task harder to accomplish.

Socket restrictions

These allow you to restrict access to client, server or both-way sockets to certain groups.

Mandatory acess control

The MAC can be used to restrict access to capabilities even to root, lessening the attack effects after they succeed.

Kernel auditing

This will enable auditing support for certain actions related to attacks (With or without the originating ip address). Specifically:

- Exec calls.*
- Resource limit oversteps.
- Exec calls inside of chroots.
- Attaching to processes via ptrace.
- Calls to chdir().*
- Mounts and unmounts.*
- Important signals like SIGSEGV which are produced by attacks oor buggy programs.
- Failing calls to fork.
- System time changes.
- ELF text relocations (useful for developers who want to avoid them).

Those marked with an * can be enabled only for a specified group.

Aftermath

After enabling the Gentoo Hardened features, the risk of the exploits aforementioned is greatly reduced, though not suppressed, as can be seen with this new table.

Anyway, it should be noted that this doesn't make updating and patching unnecessary, as there is still some risk.

Exploit	Arbitrary Code Execution	Out Of Order Code Execution	Program Flow Alteration	Denial Of Service
Executable stack overflow	0	0	0	0
Non-executable stack overflow	3	3	3	10
Mmaped space overflow	0-3	0-1	7	10
Global space overflow	1	1	5	10
Heap space overflow	1	1	5	10
User space data pointer dereference	0	0	10 (0 on x86)	7
User space function pointer dereference	0	0	0	10

Incompatibilities

Although some programs may be incompatible with some of the hardening features, the conjunction of Grsec's MAC with PAX flags and the ebuild flag filters, allow to enable and disable most of these features on a per binary circumventing most of these known Incompatibilities. Anyway it is recommended reading the help provided with the kernel options to avoid some known problems.

Thanks

I'd like to thank Magnus Granberg (Zorry) and the rest of the Gentoo Hardened team for his help while writing this document and while I was beginning at Gentoo Hardened.

I'd also like to thank Santiago M. Mola (Coldwind) and the rest of people at Polinux for his help on my first Gentoo Installation.

I'd also like to thank the people at Sofistic for their security competitions which made me see why these things are necessary.

Thanks go too to Pavel Labushev and Daniel Kuehn for his suggestions on the Deter Exploit Bruteforcing section.

LICENSE

This document is freed under a GFDL license. Anyway, I won't mind freeing it under other licenses so if you are interested, then contact me.