Security Enhancements in Red Hat Enterprise Linux (beside SELinux)

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

December 9, 2005

Abstract

Bugs in programs are unavoidable. Programs developed using C and C++ are especially in danger. If bugs cannot be avoided the next best thing is to limit the damage. This article will show improvements in Red Hat Enterprise Linux which achieve just that.

1 Introduction

Security problems are one of the biggest concerns in the industry today. Providing services on networked computers which are accessible through the intranet and/or Internet potentially to untrusted individuals puts the installation at risk. A number of changes have been made to the Linux OS[1] which help a lot to mitigate the risks. Upcoming Red Hat Enterprise Linux versions will feature the SELinux extensions, originally developed by the National Security Agency (NSA), with whom Red Hat now collaborates to productize the developed code. SELinux means a major change in Linux and is a completely separate topic by itself.

Here, we are going to concentrate on extensions Red Hat made to the OS which increase security but are not part of, and do not require SELinux. The goal for these extensions was to have no negative impact on existing code, if possible, to work without recreating binaries, and to require minimal changes to the process of building applications. The remainder of this paper introduces three separate extensions Red Hat made. It is not meant to be a complete list but rather should serve individuals who want to increase security of the Red Hat Enterprise Linux based systems as a guideline to adjust their code and installation to take advantage of the new development. Before we start with this, some words about exploiting security problems.

2 Exploiting Security Problems

When attempting to categorize security problems, one should first distinguish between remotely and locally exploitable problems. The latter can only be exploited if the attacker already can execute code on the target machine. These problems are harder to protect against since normally[2] all of the OS's functionality is available and the attacker might even be able to use self-compiled code.

Remotely exploitable problems are more serious since the attacker can be anywhere if the machine is available through the Internet. On the plus side, only applications accessible through the network services provided by the machine can be exploited, which limits the range of exploits. Further limitations are the attack vectors. Usually remote attackers can influence applications only by providing specially designed input. This often means creating buffer overflows, i.e., situations where the data is written beyond the designated memory area and overwriting other data.

One way to avoid buffer overflow problems is to use controlled runtimes where memory access is first checked for validity. This is not part of the standard C and C++ runtime, which means many applications are in danger of these problems. Intruders can misuse these bugs in a number of ways:

- if the overwritten buffer on the stack is carefully crafted, overflow can cause a naturally occurring jump to a place the intruder selected by overwriting the original return address. The target of the return might also be in the data written onto the stack by the overflow;
- a pointer variable might be overwritten. Such a variable, if located in memory in front of the overflowed buffer, could then be referenced and maybe written to. This could allow the intruder to write a value, which might also be controlled by the intruder, into a specific address;
- a normal variable might be overwritten, altering the state of the program. This might result in permission escalation, wrong results (think transfer of money to wrong accounts), etc.

[1] "OS" as in the whole system, not just the kernel.

[2] SELinux changes this.

Although these possible effects of an overflow might seem nothing but a good way to crash the application, attackers often find ingenious ways in which the application does not crash, but instead does something to the attacker's liking. This is aided by the fact that identical binaries are widely available and used in many places. This allows the attacker to study the binary locally before the attacks. Randomness in the binaries would be beneficial but for various reasons it is unpractical that end users recreate the binaries which differ sufficiently. For one, vendors will violently protest since it throws attempts to provide service and maintenance completely off the track.

3 The Plan

The best protection against these problems is, of course, auditing the code and fixing the problems. Given sufficient resources these will have some positive effect, but being human, programmers will always miss one or another problem. And there is always new code written. Besides, the investment to audit all relevant code is prohibiting. Finally, this will not at all protect against problems in 3rd party programs.

A second line of defense is needed. The worst intruders can achieve is circumventing the security of the system. If this happens, all guards of the system have no effect. Achieving this is *very* difficult with SELinux properly set up. Each application has been assigned a domain which gives it limited capabilities. Changing domains can only be done under the control of the kernel and only in ways the system adminstrator allows. Ideally, there is no all-powerful super user anymore. As stated before, we will not discuss SELinux here.

One notch down the list of dangers is the possibility for the intruder to take over the process or thread. This requires that the intruder inject code of his own choosing into the running process and cause this code to be executed.

The next level down would be an intruder changing the behavior of an application, not to enable further security escalation, but instead to cause havoc by changing the outcome of the computation (for instance, the aforementioned transfer to a wrong account).

Since C and C++ applications are usually not written to cope gracefully with bugs, the best one can hope for is to end the infected run of the application. This is the best to do if the state of the program must be assumed to be contaminated. If the application provides mechanisms to shut down only one thread or perhaps drop only the client connection which caused the problem, this can be done as well. But this requires highly sophisticated programming which is found only in applications with extreme uptime or security requirements. If the application is shut down, the worst possible outcome is that the intruder attacks again and again, causing one shutdown after the other. This can be prevented with higher-level security, by configuring the firewall or protocol analyzers.

The mechanisms developed by Red Hat automatically cause many of the common exploits to fall only in the last, and least dangerous category.

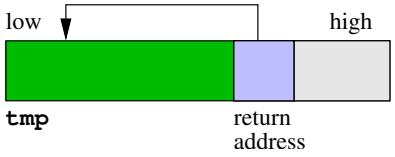
4 Exec-Shield

We start with an concrete example of some broken code which in one form or another far too often appears in deployed programs:

```
int
match (const char *passwd)
{
  char tmp[MAXPASSWD];
  if (gets (tmp) == NULL)
    return 1;
  return strcmp (passwd, tmp);
}
```

There are two factors which make this code especially dangerous. First, the use of `gets` which is *always* wrong. The function does not allow specification of the size of the buffer, so uncontrolled input can overflow the buffer. This is exactly what happens here, but what is it that is overflown? The answer is of course the array `tmp`, but there is more to it. The second point is that the array is allocated on the stack. On machines where the stack grows downward in the address space (for all common architectures other than HP/PA) the array `tmp` is located below the data generated for the function call and data the callers put on the stack. For many architectures, the data generated for the function includes the return address. Now all together this means that the input on `stdin`, read by `gets`, can be used to overwrite the return address of the function call and even set up the frame for the code to which the fake return address points. In other words, this code allows an attacker to redirect the execution of the code to an arbitrary place.

It varies where this code is that the attacker wants to execute. It could be a known address in the address space. We will handle this case in the next section. The more dangerous case is where the code comes along with the buffer overflow. Note that the standard I/O functions of the C runtime work equally well for arbitrary binary data. Often only the byte values representing newline (`'\n'`, 0x0a) and NUL (0x00) are special. All other values can be used freely. So an attacker could fill the array `tmp` with arbitrary data.



We could find ourselves in a situation as in this figure. It represents the stack memory when the example function `match` is called. The green area, the array `tmp`, is filled with data (e.g., program code) by the attacker and the return address of the function is redirected to an address inside the array `tmp`. If this return to the wrong address succeeds the intruder's code is executed. It takes less than a few dozen bytes of code to create a socket, listen on it, and redirect input and output to a shell. Voilà, a remotely accessible shell with the same privileges as the application which has been "cracked".

For this exploit to be successful a number of conditions must be met. First, the return address which will be used is an absolute address. That means if the code in the array `tmp` is to be executed, the intruder has to know the absolute address of array. The attacker can get the address wrong by a certain margin: the inserted code has to provide "landing area" which is filled with no-op operations which then permits the fake return address to be the address of any one of no-op instructions.

The second condition is that the processor has to allow execution of the code. This might seem obviously the case but it is not. Processors with virtual memory handling at some level implement a bit which determines whether the content of a memory range can be executed or not. For most architectures supported by Linux the stack has always been executable, so the exploits sketched above would work.

Removing the Conditions

The Exec-Shield extension Red Hat developed and introduced in the Fedora Core 1 release addresses these points and more. Similar and even stronger extensions existed before ([3], [4]), but neither has an effect on the execution environment low enough to be included in a general, as opposed to a high-security, Linux distribution. For instance, any restriction on the size of the available address space is completely unacceptable since there are applications which need every bit that is available. As stated before, the solutions which can be used in Red Hat Enterprise Linux must have no negative impact on existing applications. This does indeed rule out a number of drastic security measures which undeniably increase security.

The first small change Exec-Shield introduces is that the stack location is different for every process. The kernel automatically adjusts the stack address downward by a random amount of bytes. This does "waste" some memory and address space, but the possible range of the downward adjustment is chosen so that this is not a problem. This approach works since nothing in the process itself ever must depend on the exact stack address. Such a property of a process has never been guaranteed. With the stack randomization in place it is harder to create an exploit where the code loaded into memory as part of the exploit is executed.

To address the code written to the stack, the address of the stack has to be known. An attacker could potentially try several times and hope to get some kind of feedback allowing him to determine the actual address. The problem with this approach is that once the address is wrong, a jump using the address will cause execution of some arbitrary region of memory which much more often than not causes the process to crash since the memory or the code is invalid (e.g., because it is actually data). And even if the exploit can be repeated, since the process is automatically restarted, at every restart the stack address is different, so no information from the previous run can be used.

Every normally configured Linux system provides the `/proc` filesystem which exposes information about the running system. Among the information is information about each process, which in turn contains information about the memory regions in use. The file `maps` in each process' `/proc` entry shows the memory regions for the current process. This file makes locating the stack easy since the permissions allow every process to read every other process' file. The Exec-Shield therefore changes this: the `maps` file is only readable for the owner. This leaves our attacker without the necessary privileges to read this file only with the hope that due to some programming error or stupidity on the programmer's side pointer values are exposed. This should never happen and usually does not, since there is not much value. Pointers are of no use to other processes.

Removing the second factor required for the exploit requires marking the memory regions the attacker can access for writing as non-executable. In fact, the goal should be to mark as much of the address space as possible not-executable. This goal hits some problems if we do not want to change the application binary interface (ABI) or limit the user in how code can be written or what programs can be executed. On some architectures, notably IA-32, the stack is executable for a good reason: for some source code constructs the GNU C compiler (gcc) actually generates code which is written to the stack and executed there. The details are quite complicated and the feature requiring this (nested functions) is a rarely used extension gcc supports. To not prevent existing 3rd party binaries and those requiring executable stacks from running it is necessary to change the permission of the stack dynamically. The Exec-Shield extension does this by respecting information contained in the binary. The compilers and the linker were extended to keep track of whether the compiled and linked code needs an executable stack. The result is recorded in a new ELF program header entry, `PT_GNU_STACK`. The kernel uses this information to determine the initial permission. If the program, and if necessary the dynamic linker, are happy with a not-executable stack, the kernel will disallow execution on the stack. Otherwise the stack is set up for executable code. For the kernel the story ends here. But since each Dynamic Shared Object (DSO, "shared library") can bring in its own requirements, the dynamic linker

has to keep track whether there is any code loaded which needs an executable stack. When a DSO with the requirement is loaded, the permission for all stacks in use must be changed. This is plural, stacks, since the process at this point might already have created threads. The dynamic linker does all this automatically and transparently, with some help from a new kernel feature which allows to easily change the permissions of main stack. Appendix A has some more technical details on how to create binaries correctly.

Beyond the Stack

With these extensions the ability to misuse the stack is drastically reduced. But there are other parts of the address space into which the intruder could write the exploiting code and execute it. There are again two parts to this: locating the memory and executing the code. The Exec-Shield extensions try to address both.

To prevent easily locating the writable data memory, they should be placed at different addresses for every run of the process, just as it happens for the stack. The writable data memory is usually not alone, though, its position relative to the accompanying code is usually fixed. This means the entire binary must be loaded at different addresses every time. Doing this provides no problems for DSOs which are by definition freely relocatable. The kernel randomizes the addresses for requests from user-level when a mapping of a file is requested without fixed requirement on the address (i.e., the first parameter for `mmap` is `NULL`). The dynamic linker never insists on a specific load address but it can suggest addresses if the application is prelinked.[3] This means prelinking and load address randomization exclude each other.

One possibility opened by the load address randomization is that the kernel can choose to map binaries in the first 16MiB of the address space. The noteworthy aspect of this is that all addresses in this range contain a NUL byte. As mentioned above, NUL is one of the two special characters in standard I/O handling. More concrete, it is special in string handling. It is not possible to handle a copy of a string with `strcpy` or similar functions beyond the NUL byte. For the attacker, who has to insert addresses of the code which is called for the exploit, this poses a big problem if the representation of that address contains a NUL byte. This part of the address space is rightly referred to as the ASCII-armor area. By moving as much code to the first 16MiB of address space, a lot of code is out of reach for this type of attack. If there is room in the memory region, the kernel will map all memory there for which the protection bits include `PROT_EXEC`. The dynamic linker always set this bit for the first `mmap` call to load a DSO, so things happen automagically.

[3] See the `prelink` package. Prelinking is a way to speed-up program startup.

By doing all this, only one fixed rock is left in the address space: the executable itself. An executable, as opposed to DSOs, is linked for a specific address which must be adhered to, otherwise the code cannot run. Red Hat developed a solution for this problem as well, which will be addressed in the next section. The executable itself is a bit special though. for it not only consists of the usual code and data parts, but it also has the `brk` area attached to it. The `brk` area (aka heap) is a region of the address space in which the process can allocate new memory for interfaces like `malloc`. This area started traditionally right after the BSS data of the executable (BSS data is the part of the data segment which holds the uninitialized data or the data explicitly initialized with zero). But there never has been any formal specification for this. And in fact, since almost no program (for good reasons) uses the `brk` interfaces directly, user programs are never exposed to the exact placement of the heap. Which brings us back to randomization: the Exec-Shield patch randomizes the heap address as well. A random sized gap is left between the end of the BSS data and the start of the heap. This means that objects allocated on the heap do not have the same address in two runs of the same program. The change has remarkably little negative effects. The only known problem is that while building emacs (not running the final program) the dumping routines depend on the heap directly following the BSS data. The workaround is to disable Exec-Shield temporarily.

Implementation Details

In the discussion of Exec-Shield so far, we glossed over the implementation details. Exec-Shield is a kernel patch and all the changes happen transparently for the user. But one detail of the implementation is worth noticing.

The developers of the 80386, the first implementation of the IA-32 architecture with protected mode, saved some precious chip real estate by not implementing the flag governing execution permission for each memory page in the virtual address space. Instead, the permission for 'read' and 'execution' are collapsed into one bit. Without making data unreadable, it is not possible to control execution this way.

There is a way to control execution, though, but it is convoluted. The segmentation mechanism of the processor provides a coarse mechanism to control execution. Without the Exec-Shield patch, the segment used for program execution (selected with the `%cs` register) stretches the whole 4GiB and therefore contains the stack and all data. The Exec-Shield patch changes this. The kernel now keeps track of the highest address which has been requested to be executable. All addresses from zero to the highest required address are kept executable. Requests for executable memory are made exclusively by calls to `mmap` or `mprotect` and the implicitly added mappings of the program itself and the dynamic linker. This means the stack is usually not executable. Since new mappings

with the PROT_EXEC bit set are mapped into the ASCII-armor area, but pure data mapping are mapped high up, this means the range of executable code is kept minimal and data usually is not executable. If an intruder has control over the application this protection can easily be defeated by calling mprotect with a PROT_EXEC parameter for an object high up in the address space. But the Exec-Shield patch is about preventing the intruder to get such control, not to contain him afterward.

5 Position Independent Executables

In the previous section it has been described how the Exec-Shield patch makes an attacker's life harder by randomizing the addresses where various parts of the running program are located. With one exception: the executable itself. There is nothing the kernel can do to change this. But the programmer can.

To load an executable at different addresses every time it must be built relocatable. This sounds familiar: a DSO is relocatable. Therefore, Red Hat modified the compiler and linker to create a special kind of executable: Position Independent Executables (PIEs). PIEs are a merger between executables and DSOs. From the kernel's point of view PIEs are nothing but DSOs. The Linux kernel for a long time supports executing DSOs just as if they were executables so no kernel changes are needed.

The tools had to be extended, though. Normal DSOs do not contain all the information an executable has and, equally important, are compiled in a more general way which makes the code slower than necessary. But one step at a time.

To create a PIE, the compiler and linker need to be told about this. The compiler has two new options, -fpie and -fPIE which are analogous to the already present -fpic and -fPIC options. Just like the counterparts, the two new options instruct the compiler to generate position independent code. But on top of this, the compiler can assume that all symbols defined locally are also resolved locally. A detailed explanation of this is long and complicated. The interested reader is referred to [1].

When generating PIEs it is important to ensure all object files linked into the application are position-independent. For the application itself it means that all files should be compiled with the -fpie option (or -fpic, though less optimal). The more dangerous part are files linked in from archives, especially when they come from archives which are not part of the program's package itself. The files in the archive must also be position independent which might require coordination with another package. Additionally, if the archive is also used for other purposes, compiling the contained files with -fpie might actually be wrong. If the code ends up in a DSO the symbol resolution rules would be violated.

Once all the files are compiled, the linker has to be told that a PIE has to be created. This works by adding the -pie option to the command line. The gcc driver program then makes sure the correct crt files are linked in etc. Needless to say that the -fpie option as well as -pie are not supported if an old version of gcc or a non-gcc compiler is used.

It is easy to miss a position-dependent file linked into a PIE. Therefore, developers should always check afterward whether the PIE is free of text relocations. Text relocations are the result of such position-dependent code being used. The linker will detect the problems, though, and add a flag to the PIE's dynamic section. One can check for the flag with this:

```
$ eu-readelf -d BINARY | fgrep TEXTREL
```

If this pipeline produces any output, the program contains text relocations and should be fixed. It is not necessary for correct execution, but running a program with text relocation means the memory pages affected by the relocations are not sharable (increasing resource usage) and that startup times can be significantly higher. Text relocations also can create security problems since otherwise write-protected memory briefly becomes writable. The script in appendix B will flag PIEs with text relocations, among other things.

Many applications which are directly exposed to the Internet and some other security relevant programs are converted to PIE in Red Hat Enterprise Linux and Fedora Core. It does not, in general, make sense to convert all binaries. Running PIEs is excluding them from taking advantage of prelinking. The kernel and dynamic linker will randomize load addresses of all the loaded objects for PIEs with the consequence that the PIEs start up a bit slower. If startup times are not an issue (and we are talking about differences usually in the sub-second range, often much lower) PIE can be used freely. All long-running daemons are good candidates and certainly all daemons accepting input from networks. But also applications like Mozilla, which can be scripted from the outside, should be converted.

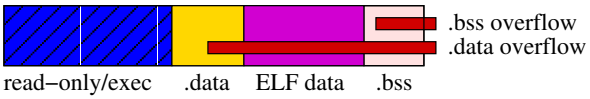
6 ELF Data Hardening

With Exec-Shield and PIEs we have done work on the big building blocks of a running application. After this it was time to look at the individual blocks in detail to see what can be done to increase security at that level. The individual files are all ELF files which, looked at in more detail, present themselves as a sequence of sections which each have a certain purpose. The following list shows the various sections in a normal IA-32 application in the order a linker would create so far.

```
[ 1] .interp               PROGBITS
```

```
[ 2] .note.ABI-tag        NOTE
[ 3] .hash                HASH
[ 4] .dynsym              DYNSYM
[ 5] .dynstr              STRTAB
[ 6] .gnu.version         GNU_versym
[ 7] .gnu.version_r       GNU_verneed
[ 8] .rel.dyn             REL
[ 9] .rel.plt             REL
[10] .init                PROGBITS
[11] .plt                 PROGBITS
[12] .text                PROGBITS
[13] .fini                PROGBITS
[14] .rodata              PROGBITS
[15] .eh_frame            PROGBITS
[16] .data                PROGBITS
[17] .dynamic             DYNAMIC
[18] .ctors               PROGBITS
[19] .dtors               PROGBITS
[20] .jcr                 PROGBITS
[21] .got                 PROGBITS
[22] .bss                 PROGBITS
[23] .shstrtab            STRTAB
```

The first 15 sections do not have to be modified at runtime and can be mapped into memory to not allow write access. The remaining section, except number 23 which is not needed at runtime at all, are data sections and need to be modified. This is the part of the program which is putting the program in danger. Any place which is writable is a possible target for an attacker.



This graphic shows the different parts of the ELF file. The hatching indicates the memory is write-protected. The red bars indicate which areas a potential buffer overrun in the `.data` and `.bss` section respectively can easily affect.

For instance take the `.got` section. This section (part of the violet colored area) contains internal ELF data which is used at runtime to find the various symbols the program needs. The section contains pointers and the pointers are simply loaded from that section and then dereferenced or even jumped to. An attacker who could write a value to this section would be able to redirect the data accesses or function calls done using the entries of the `.got` section. Other sections fall into the same category. There are actually only two real data sections the program uses: `.data` and `.bss`. Note that the `.rodata` section containing truly read-only data, like constants or strings, falls into the aforementioned 15 sections. And even this is not the entire story. Consider the following code:

```
const char *const msgs[] = {
  "message one", "message two"
};
```

The array `msgs` is declared `const` but in a position independent binary, the addresses of the strings that the elements of the array point to are not known at link-time. Therefore, the dynamic linker has to complete the relocation by making adjustments which take the actual load address into account. Making adjustments means the content of the array `msgs` has to be writable. This is why in the section layout above the array `msgs` would be placed in the `.data` section.

Even though this is what the linker does, this is not the optimal result. The compiler actually does better. It emits the array in a separate section named `.data.rel.ro` which contains data that needs to be modified by relocations, but is otherwise read-only. Unfortunately there is no match for this in the current section layout.

This is not the worst problem, though. The order in which the writable sections are currently lined up has only historic reasons, not technical ones. Unfortunately, not much thought went into the layout so far. If an array in the `.data` section is overflown, it is possible to modify all of the following section, especially including the `.dynamic` and `.got` sections. This is something which in many situations can be avoided by simply reordering the sections so that the sections with ELF data structures precede the program's data sections. This does not mean that overwriting the program's data is not harmful and cannot be exploited, but protecting the ELF data structures removes yet another weapon from the arsenal of the attackers. The IA-32 binutils package available in Fedora Core 2 and later releases by Red Hat would produce the following section layout:

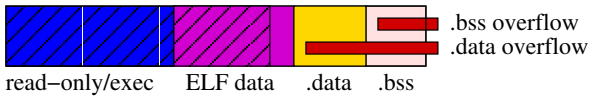
```
[ 1] .interp              PROGBITS
[ 2] .note.ABI-tag        NOTE
[ 3] .hash                HASH
[ 4] .dynsym              DYNSYM
[ 5] .dynstr              STRTAB
[ 6] .gnu.version         GNU_versym
[ 7] .gnu.version_r       GNU_verneed
[ 8] .rel.dyn             REL
[ 9] .rel.plt             REL
[10] .init                PROGBITS
[11] .plt                 PROGBITS
[12] .text                PROGBITS
[13] .fini                PROGBITS
[14] .rodata              PROGBITS
[15] .eh_frame            PROGBITS
[16] .ctors               PROGBITS
[17] .dtors               PROGBITS
[18] .jcr                 PROGBITS
[19] .dynamic             DYNAMIC
[20] .got                 PROGBITS
[21] .got.plt             PROGBITS
[22] .data                PROGBITS
```

```
[23] .bss                  NOBITS
[24] .shstrtab             STRTAB
```

The first 15 sections have not changed and we can ignore the last section since it is not used at runtime. The data sections have changed drastically. Now all the sections with ELF internal data precede the program's data sections `.data` and `.bss`. And what is more, there is a new section `.got.plt` whose function is not immediately apparent. To take advantage of this additional section one has to pass `-z relro` to the linker (i.e., add `-Wl,-z,relro` to the compiler command line). If this is done the ELF program header gets a new entry:

```
eu-readelf -l BINARY | fgrep RELRO
  GNU_RELRO      ...
```

This entry specifies what part of the data segment is only written to during the relocation of the object. The intent is that the dynamic linker marks the memory region as read-only after it is done with the relocations. The dynamic linker in glibc 2.3.4 and later does just that. We get the following changed picture:



We see the enlarged write-protected area and the buffer overruns can 'only' affect the `.data` and `.bss` sections easily.

To enable changing the permission in the data, the linker has to add some padding on the file. Memory page permission can only be changed with page granularity. This means that if a page contains just one byte which needs to be written to, it cannot be marked as read-only. The linker therefore aligns the data so that the data which is read-only after relocation is on a separate page after loading the data. This is why we now have the separate `.got.plt` section: the first part of the Global Offset Table (GOT) is modified only during the relocation. The second part, associated with the Procedure Linkage Table (PLT), is modified later as well. It is therefore kept along with the program's data in the part of the data segment for which the protection is not changed.

One tiny detail: it is not entirely true that the `.got.plt` section is always modified after relocation. In case no runtime relocation happens this is not the case. And the programmer can enforce this by adding the `-z now` linker option. If this option is used, the linker sets a flag in the generated binary which causes the dynamic linker to perform all relocations at startup time. This slows down the startup, in some cases significantly, and might in some very rare cases even alter the behavior of the application. But the benefit is that the linker can move the `.got.plt` section also in the region, which is read-only after the relocation. This is good protection, since known attacks do target this part of the GOT. Daemons which are long-running and especially endangered networked application should be linked with `-z now` to add the extra protection.

Upcoming Red Hat Enterprise Linux releases will have all applications created with the new linker which orders the sections correctly. In addition, each program is examined whether it is a candidate for the addition of the `-z relro` and `-z now` option. After all this protection is applied, the only memory an attacker can write to is the stack, the heap, and the data sections of the various loaded objects. And unless there are good reasons, none of these memory regions is executable.

7 Conclusion

These security enhancements described in this paper make noticeable impact on known exploits. They do not, however, prevent the exploitable program bugs in the first place. These are still present and attacker can take advantage of them. The changes do often radically reduce the consequence. Instead of being remote root shell attacks program bugs often are mere Denial of Service (DoS) attacks. These are not nice and disturb a systems operation but they do not necessarily mean security problems and they are easier to handle. System monitoring software can detect a program crashing and it can keep track of this. If crashes are suddenly frequent system administrator can be alerted to the fact.

Together with the SELinux integration into the Linux kernel these changes make the life of intruders harder. No program is disrupted, if a program is not adjusted for the security enhancements it will continue to work as before. There are no restriction of the use of the virtual address space which together means that the resistance to introduce these features is minimal. Disruptions are still possible, but the severity of the attacks is significantly reduce which will make system administrators and legal departments very happy.

A Using Exec-Shield

The GNU C compiler and the linker usually determine whether the code needs an executable stack correctly. To see what is recorded one can run commands like these:

```
$ eu-readelf -l /bin/ls | fgrep STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x000000 0x000000 RW  0x4
```

The second to last column of the output shows that the stack for `ls` need not be executable, only read-writable (`RW`). If the output is `RWX` the binary is marked to need an executable stack and the kernel or dynamic linker will make the necessary adjustments. In any case, this is a sign that one should examine the binary since it might be unintentional.

Unintentional execution permission can be granted if any files linked into the binary were compiled with a compiler which does not add the necessary attribution of the object files, or the file was written in assembler. In the former case one must update to more recent versions (in case the compiler is a GNU compiler) or demand from the vendor that the necessary instrumentation is done. The linker always defaults to the safe side: if any input file does not indicate that a not-executable stack is OK, the resulting binary will be marked as requiring an executable stack.

The case of assembler files is more interesting since it happens even with an up-to-date GNU compiler set installed. There is simply no way the assembler can determine the executability requirement by itself. The programmer must help by adding a sequence like the following to every assembler file:

```
  .section .note.GNU-stack,"",@progbits
```

Alternatively, the GNU assembler can be told to just add this section, regardless of the content of the file. This is possible by adding `-Wa,--execstack` to the compiler command line. Note this will not work if an alternative assembler like nasm is used. For nasm, add the following line to the input file (probably as the last line in the file):

```
  section .note.GNU-stack progbits noalloc noexec nowrite align=1
```

Once the binary is created, the information needed to make a decision is usually lost. If a user knows for sure that no executable stack is needed, it is often possible to mark the finished binary appropriately. This is especially useful for binaries, executables and DSOs, which are not available in source form. The tool to use is called `execstack` and it is part of the `prelink` package. Running

```
$ execstack -s /usr/java/*/bin/java
```

adds a `PT_GNU_STACK` entry in the program's program header. Adding this entry might sometimes fail for executables. Adding something in the middle of the executable cannot work if not all the interdependencies between the different pieces of the executable are known. Those interdependencies are lost at link-time unless the `-q` option for the linker is used.[4]

There is one more way to influence the stack handling. The kernel allows the system administrator to select a global setting influencing the use of Exec-Shield. One of three variants can be selected by writing one of the strings `0`, `1`, or `2` into the file `/proc/sys/kernel/exec-shield`:

`0` Exec-Shield is completely disabled. None of the described protections is available in any process started afterward.

`1` The kernel follows what the `PT_GNU_STACK` program header entry says when it comes to selecting the permissions for the stack. For any binary which does not have a `PT_GNU_STACK` entry, the stack is created executable.

[4] This option is rarely used and chances are it does not work.

2 This option is similar to `1`, but all binaries which do not have a `PT_GNU_STACK` entry are executed without executable stack. This option is useful to prevent introducing problems by importing binaries. Every unmarked binary which does need an executable stack would have to be treated with `execstack` to add the program header entry.

For debugging purposes it might be useful to not have the load address of binaries, DSOs, and the address of the stack at a different place every time the process is restarted. It would be harder to track variables in the stack. To disable just the stack randomization the system administrator can write `0` into `/proc/sys/kernel/exec-shield-randomize`.

B Script to Test for Safe Programs

It is nice to have all these security improvements available. But how can one be sure they are used? Red Hat uses the following script internally which checks currently running processes. Output can be selected in three different ways. For each process, the script prints out whether the program is a PIE and whether the stack is writable or not. Especially the later output is useful since no static test can be as thorough. At runtime, the permissions can change and this would not be recorded in the static flags. Every process marked to have Exec-Shield disabled is a possible problem. If the stack is executable just because the flag is missing, use the `execstack` tool (see previous section). If a program is shown to not be a PIE this does not necessarily mean this is a problem. One has to judge the situation: if the process is a high-risk case since it is accessible through the network or is a SUID/SGID application, it might be worth converting the application into a PIE.

```
#!/bin/bash
# Copyright (C) 2003, 2004 Red Hat, Inc.
# Written by Ingo Molnar and Ulrich Drepper
if [ "$#" != "1" ]; then
  echo "usage: lsexec [ <PID> | process name | --all ]"
  exit 1
fi
if ! test -f /etc/redhat-release; then
  echo "this script is written for RHEL or Fedora Core"
  exit 1
fi

cd /proc

printit() {
    if [ -r $1/maps ]; then
     echo -n $(basename $(readlink $1/exe))
     printf ", PID %6d: " $1
     if [ -r $1/exe ]; then
       if eu-readelf -h $1/exe|egrep -q 'Type:[[:space:]]*EXEC'; then
         echo -n -e '\033[31mno PIE\033[m, '
       else
         if eu-readelf -d $1/exe|egrep -q '  DEBUG[[:space:]]*$'; then
           echo -n -e '\033[32mPIE\033[m, '
           if eu-readelf -d $1/exe|fgrep -q TEXTREL; then
             echo -n -e '\033[31mTEXTREL\033[m, '
           fi
         else
           echo -n -e '\033[33mDSO\033[m, '
          fi
       fi
       if eu-readelf -l $1/exe|fgrep -q 'GNU_RELRO'; then
         if eu-readelf -d $1/exe|fgrep -q 'BIND_NOW'; then
           if eu-readelf -l $1/exe|fgrep -q ' .got] .data .bss'; then
             echo -n -e '\033[32mfull RELRO\033[m, '
           else
             echo -n -e '\033[31mincorrect RELRO\033[m, '
           fi
         else
           echo -n -e '\033[33mpartial RELRO\033[m, '
         fi
       else
         echo -n -e '\033[31mno RELRO\033[m, '
       fi
     fi
     lastpg=$(sed -n '/^[[:xdigit:]]*-[[:xdigit:]]* rw.. \
                     \([[:xdigit:]]*\) 00:00 0$/p' $1/maps |
      tail -n 1)
     if echo "$lastpg" | egrep -v -q ' rwx. '; then
       lastpg=""
     fi
     if [ -z "$lastpg" ] || [ -z "$(echo $lastpg||cut -d ' ' -f3|tr -d 0)" ]; then
```

```
       echo -e '\033[32mexecshield enabled\033[m'
     else
      echo -e '\033[31mexecshield disabled\033[m'
      for N in $(awk '{print $6}' $1/maps  | egrep '\.so|bin/' | grep '^/' \
                     | sort -u); do
        NE=$(eu-readelf -l $N | fgrep STACK | fgrep 'RW ')
        if [ "$NE" = "" ]; then
          echo " => $N disables exec-shield!"
        fi
      done
     fi
    fi
}

if [ -d $1 ]; then
  printit $1
  exit 0
fi

if [ "$1" = "--all" ]; then
   for N in [1-9]*; do
     if [ $N != $$ ] && readlink -q $N/exe > /dev/null; then
       printit $N
     fi
   done
   exit 0
fi

for N in $(/sbin/pidof $1); do
 if [ -d $N ]; then
  printit $N
 fi
done
```

C References

[1] Ulrich Drepper, *How To Write Shared Libraries*, http://people.redhat.com/drepper/dsohowto.pdf, 2003.

[2] Ingo Molnar, Announcement of Exec-Shield, http://people.redhat.com/mingo/exec-shield/ANNOUNCE-exec-shield, 2003.

[3] grsecurity Linux kernel patches, http://www.grsecurity.net/features.php.

[4] PaX, http://pageexec.virtualave.net.

D Revision History

2004-1-22 First internal draft.

2004-1-23 Fix typos. Add graphics for ELF file layout.

2004-2-2 Word tweaking.

2004-2-23 Some more typos fixed. Little clarifications.

2004-2-29 More typo fixes.

2004-5-21 Fix typos. Explain BSS a bit. Reported by Mark Cox.

2004-6-16 More typo and language fixes. By Kristin Horne.

2004-6-26 Yet more typos. By Neal Lao.

2005-2-23 Document how to make Exec-Shield work with nasm.