Safe Programming Practices

Introduction

In recent years as the Internet has become increasingly popular, the number of threats on the Internet has grown dramatically. Many of the threats such as viruses and trojans often exploit already existing programs on the victims computer which are faulty in some way or the other, making it possible for the attacker to install his malicious program on victims computer which could take complete control over the victims computer without him knowing it. A consequence of this development has been that its necessary to check for software updates at least on a weekly basics, which is not something the average user would necessary do for software which doesn't check for updates itself. It would clearly be easier to educate the developers to write more secure programs than to educate all the users to keep up to date with all the latest security issues, trends and buzz-words. There are some common pitfalls for developers, which they may not know about or simply have overlooked due to other factors such as bugs or deadlines. I have taken a look at the common pitfalls and potential dangers which could occur when developing software and will go through some of most popular types of code exploitation and look at methods of prevention.

Unsafe code

All programs (with very few exceptions) take input and most of these directly from the user or user files. If the program does not validate the input throughly, malformed data could pass all the checks and cause the program to behave different than what was intended, changing the flow of the program.

Another possibility for passing malformed data is that it constructed in such a way that it overflows a buffer and overwrites unrelated data. This unrelated data could be an address pointer the CPU will be executing instructions at in the future, making it possible to insert malicious code in the buffer that is being overflowed and by making the address pointer point to the beginning of the buffer, this would yield control over the CPU when it executes the instructions the address pointer points to. Lets take a closer look at this two scenarios, what the consequences would be in the worst case and what could be done to prevent it.

Changing program flow

Even though changing the flow of a program does not seem as critical and serious as taking over CPU execution, the consequences for a company can be just as fatal. Consider a website on the Internet, made in some sort of scripting language, which takes parameters are appended to the end of the URL, like "http://www.mywebsite.com/section.php?s=about_me.dat". In this example the s= parameter tells the website which file it should display content from. One can imagine that if this is not properly checked or the web server is not configured correctly, this could lead to reading any file on the server. In a danish newspaper they reported on the front page about a dating site on the Internet where you could access private sections of a users profile, which you would normally only be allowed if the owner of the profile granted you access. The newspaper published private pictures of users from the website in the newspaper, thus giving the website a bad reputation.
Such programming errors are easy to track within the source code directly by passing the malformed URL directly and letting program print debug information.
It is difficult to prevent such errors as programs are often very complex and human error does occur. One way prevent such errors, a group of developers could review the code and try to come up with malformed data which could possibly compromise the the system. Its often hard for developers to spot their own errors, therefor it is important that the code gets reviewed by a group which was not involved with programming the system and thus increasing the chances of finding bugs.
In general one should just be handle invalid input just as carefully as one handles valid input.

Taking over execution

Even though getting access to private data by malformed data is sometimes all you want as a hacker, it is far more interesting if you could get access to a system and even more interesting as a superuser, thus giving complete control over the system. Like mentioned previously one could take control over the CPU by overflowing a buffer and overwriting an instruction pointer, in reality this buffer is the stack and the address pointer you overwrite is the return address on the stack frame. This is of course only possible in such programming languages as C/C++/Pascal/etc. where you can access any memory address not protected by the operating system directly.
Thanks to the current computer architecture the stack grows upwards, which means that all stack

frame data is below the user data on the stack, which means that if you can overflow a buffer, you can overwrite anything in the stack frame, including the return address for a function. The most common way to overflow a buffer is by taking advantage of how ASCII strings are represented. A string is terminated by the ASCII value 0, often called NULL[1]. I the C programming language there are string functions in the standard C library which do not terminate until they reach a NULL. For instance strcpy (string copy), which copies a string from one buffer to another, it will copy bytes from the source to the destination until it reaches a NULL. If this source buffer is user input and the destination buffer is a local buffer in a function with a fixed size, then this buffer will reside on the stack and thus a malformed string provided by the user can be longer than the buffer and thereby overflow it. If the string is carefully constructed you could overwrite the return address on the stack with the address where the buffer starts and have code inside the buffer.

Currently neither Linux or Windows distinguishes between code and data segments, so its possible to execute instructions in a data segment, afact that makes buffer overflows possible.

The code you put inside the buffer would in Unix systems be a system call to execute a command shell such as bash. If the program you tried to overflow was running in supervisor mode and this program then would execute a shell, you would be running a shell in supervisor (root) mode and thus having complete control over a system. Many popular software packages such as wu_ftp, bind and apache just to name few have been exposed to these kind of buffer overflow attacks.

Spotting such potential overflows in a program just by going through the source code is not always easy as it requires some knowledge regarding the computer architecture you are programming on and potential dangers of programming language your are using. It seems to be a somewhat “black magic” for average programmers how exploits are actually made and documentation on how make them are often written by hackers who hide behind a nickname which makes the whole thing seem like an underground hacker thing. I however, believe that every programmer should be taught how to write the most common type of exploits and get some hands-on experience, because if they can write an exploit they can also prevent one.

There has been done some work to make it much harder to make buffer overflows in some operating systems such Linux, for which there are patches available for the 2.6.x kernel which randomizes the stack pointer on every execution of a process. One of the already difficult parts of making an exploit work is that the return address you are overwriting is an absolute address, so you need to know the

1 NULL can also refer to the memory address 0

exact address of where the buffer you are overflowing starts. This can be done relatively easy if the stack pointer is static, you can brute force it starting from where the stack pointer is pointing at the beginning of a process and work your way up through memory. You can also do some tricks such a adding a lot of NOPs (no operation) in beginning of your overflow, making it more likely that you will find a address from brute forcing, as there is a larger memory area which is a valid entry point for the exploit. However with a randomized stack pointer you do not have an starting address you can brute force from nor can you compare results from several runs. It is likely that we will see this kind of feature in other operating systems as well as separation of code and data, making it impossible to execute code from a data segment.

As a developer you need to keep up with whats currently happening security wise in the programming languages you work with and follow guidelines for writing safe code.

Conclusion

After taking a closer look at some of the common programming pitfalls, it is more clear how important it has become to write safe code as there are people who might try to exploit your code, be it for fun or for profit.

References

"Exploiting Format String Vulnerabilities" - scut / team teso

"Smashing The Stack For Fun And Profit" - Alephe One

http://doc.bughunter.net