Securing Software:
The Methods, the Problems, the Solutions?

Mike Gualtieri
mikeg@cs.pitt.edu
Department of Computer Science
University of Pittsburgh

Abstract

All software has the potential to contain vulnerabilities. Currently a common practice is to "hope and pray" that a system won't become compromised until a vendor releases a patch to correct the problem. Several packages have been released which attempt to protect a vulnerable system from becoming compromised, despite having vulnerable software. These methods have weaknesses and strengths, which must be evaluated before attempting to solve this problem.

1.0 Introduction

Thought by many as only a hacking/cracking e-zine, read by people in the dark corners of the internet, Phrack published an article in 1996, which changed the face of software security. "Smashing The Stack For Fun And Profit" [9], written by "Aleph One", the former maintainer of the bugtraq security list, quickly became one of the most important articles on the topic of locating and exploiting buffer overflows in software. This once coveted knowledge was now brought into the forefront for everyone to learn.

Since 1996 countless buffer overflows have been found in various software on all platforms/architectures, and continuously plague new software. Buffer overflows, however, aren't the only problems lurking in software. New types of attacks are being discovered and exploited continuously. Format string violations, malicious Java script/Active X execution in web browsers, race conditions, to name a few, all can lead to the compromise of a system. That list doesn't even include problems from virii and trojans, which effect internet users continuously.

New software is still plagued by these same problems. At the years end in 2000, Microsoft Windows NT 4.0 had a total of 71 vulnerabilities and RedHat Linux 6.2 (i386) had 65. Windows XP Home/Professional has already had 17 reported vulnerabilities, since its recent release [10, 11].

Waiting for vendors to patch software isn't always an option, and once the patch is released will it affect another part of the system? What about the vulnerabilities that are not reported, those used for fun and/or profit such as Aleph's article suggests? Patches can only be created if the vendor is aware of the problem.

A better solution is to build a system that mitigates vulnerabilities, despite having vulnerable software. Several attempts have been made within the last few years that try to accomplish such a feat. Several dynamic translation packages look into the runtime environment of a currently running program, to detect problems. Other static analysis packages attempt to weed out vulnerabilities before execution or at compile time. However there isn't one package that does it all, and improvements can still be made.

2.0 Types of Threats

There are many ways software can be attacked. Buffer overflows and format string violations are amongst the most common ways to compromise a particular software to elevate privileges on a system, however, certain conditions must be present to do so.

On UNIX based systems, programs of interest are those that are `setuid(0)`, those which are executed with root level privileges. This is a commonalty in UNIX systems, because a user may not have privileges to perform certain operations, however may run a program that executes these operations for them.

One method to combat setuid problems is to start the program in a `chroot` environment, or basically change the location of the root directory. This is a safety precaution, so if a compromise occurs, the attacker will have limited access to which files can be viewed. The key thing to note here is that a compromise may still occur. Setting up a secure environment for every program running on a system can be complex, and where there is complexity there is room for mistakes. In sections 3 and 4 different methods for dealing with threats will be examined. In this section, certain types of attacks will be looked at more closely.

2.1 Buffer Overflows

A buffer overflow occurs when more data than a certain buffer can handle is put into the buffer. Here is an example [9]:

```
void function(char *str)
      { char buffer[16];

        strcpy(buffer,str);
      }

void main()
      { char large_string[256];
        int i;

        for( i = 0; i < 255; i++)
        large_string[i] = 'A';

        function(large_string);
      }
```

Here `strcpy`, which does not do bound checking, copies an array of the size 256, into the allocated space of only 16. The first 240 bytes on the stack are being overwritten, in this case with the letter 'A', or 0x41 in hex. When executed this will cause a segmentation fault. The problem here occurs because the return address of the function has been changed to 0x41414141. Without getting into the gory details, it is possible to input shell code, code which spawns a shell on the system, padded with data into a buffer, then return to the start of the shell code when it overflows. If the program is running as `setuid(0)`, then a rootshell is gained.

2.2 Format String Violations

A format string violation occurs if a user can pass string formats to a program. For example, by executing:

```
scanf("%s", str);
printf(str);
```

We can input print formats into `scanf (such as %x)`, which are then executed with the `printf` statement. The problem occurs when an attacker passes a %n to the string, which stores in a pointer the number of characters output in the current printf statement. This could allow an attacker to store an arbitrary value in any memory location, such as a return address of a function. Once the function attempts to return, it will jump to and execute the attackers code.

For example [12]: If exploit code was located at 0x0012FF40 we would need the `printf` to format 0x0012FF40 bytes. The resulting statement, to be passed into the `printf`, would be `%.622496x%.622496x%n`.

This would not directly cause an exploit, however, if shell code was added to the beginning of the statement, we could execute it.

For example[12]: The string: `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%.622496x%.622456x%n`, when passed into `printf` would cause the code AAA...AA to be executed. While this does nothing, actual exploit could be substituted. Note that we must subtract the length of the shell code from the second memory location.

2.3 Polymorphic Virii

In addition to traditional software compromise, virii inject themselves into programs/files and continue to execute and spread. Most virii can easily be eradicated by anti-virus (AV) programs. Currently a common method of detection, is to look for certain strings which are unique to a virus. Once found, the virus is eradicated.

Polymorphic virii pose a more serious problem for AV programs. These virii continue to change and modify themselves to avoid detection. A common method for the execution of a polymorphic virus is as such. The virus body is encrypted, so no recognizable string can be found. The only part of the virus that is unencrypted at all times is the decryption engine, whose job is to decrypt the virus, so it can execute. In addition to the decryption engine is the mutation engine, whose job is to continuously change the code of the decryption engine so it executes different instructions every time, however produces the same results.

This poses a unique problem to AV programs, because simple search strings cannot be used for the detection of a virus. Certain method can be used to detect such a virus, which will be discussed later.

3.0 Statically Analyzing Software for Vulnerability

By statically analyzing the source code of a program, its vulnerabilities may be discovered. In this section several methods to accomplish this will be discussed, ranging from static code analysis to compiler extensions to provide security.

3.1 Statically Detecting Likely Buffer Overflow Vulnerabilities [3]

In this paper a tool is described which they built to statically check programs for buffer overflows. Their tool is built on LCLint, which is "an annotation-assisted lightweight static checking tool." LCLint uses annotations within the source code, provided by the programmers, to check assumptions the programmer made. When an assumption is broken such as "notnull" LCLint will return a warning.

Using this functionality, they built extensions onto LCLint to perform security checks. One such extension was to check `strcpy`, a function notorious for buffer overflows, with a max buffer size.

To implement these checks they combined traditional compiler data flow analysis with constraint generation and resolution. The constraints used were `maxSet, minSet, maxRead, and minRead`. Using these constraints in combination with control flow analysis, they were able to see if violations could possibly occur, when the program was executed.

They tested their tool against wu-ftp-2.5.0 and BIND-8.2.2p7, with useful results. In both applications they were able to find several problems, some leading to buffer overflows. Their tool also ran fairly quick, comparing to the time of a compiler.

However, the tool has drawbacks. Many false warnings occurred while testing, (in the case of wu-ftp out of 143 warnings 88 were false). Also, it does not catch every overflow problem. The biggest disadvantage is that it depends on annotations to be provided in the source code. While no modifications to the code need to be made, the annotations must be put in place unless already there.

3.2**StackGuard: Automatic Adaptive Detection and prevention of Buffer Overflow Attacks** [5]

StackGuard is a compile time tool which "virtually eliminates" buffer overflow vulnerabilities in software. It is implemented as a patch to gcc, and requires no source code changes. When a program falls victim to attack, instead of giving control of the system to an attacker, it falls into a fail-safe mode, to prevent further damage.

It works by detecting if a return address has changed from a function. This however must be done before the return has been made. What StackGuard does is place a *canary* word in front of the return address. If the canary is intact, then the return call is made. The drawback here is that StackGuard can be evaded if the canary word is detected. Also, in some cases it is possible to skip over the canary word, however this is extremely difficult.

Because the canary method may sometimes be overcome, a more secure environment can be gained with MemGuard. MemGuard is a tool, which can protect individual words in memory. This can be used to prevent buffer overflow attacks by protecting the return address of a function when it is called, and un-protecting it when it returns. However, the drawback is that it is slower.

To get the best of both worlds, namely speed and security, they propose a hybrid approach. The system uses the canary approach most of the time, unless an attack is detected within the system, in which case the MemGuard approach kicks in and elevates security. This also adds the improvement of being able to recover from an attack. Another advantage is that, with the canary approach, when an attacker exploits a vulnerability the process is halted. With the hybrid approach, when MemGuard is activated, the process can recover and continue to execute, slower but more secure.

3.3 **Intrusion Detection via Static Analysis** [7]

By statically analyzing source code, a model of a program's behavior can be created. If the program's flow of execution is altered from this model then something has went awry, and most likely an attack is taking place.

They make the assumption that an attack cannot cause much damage if it isn't allowed to interact with the operating system, such as StackGuard's approach. The only way for a program to interact with an operating system is through system calls, so thus only system calls need to be modeled. By removing the complexity of the source code, and only modeling the system calls, a much quicker model can be generated. This model is then simulated in an interpreter along side the application, to check for proper transitions of the source code.

This system tends to work fairly well, however it does not defend against "mimicry attacks", or attacks that look to the model like the same code is executing.

4.0 Dynamically Analyzing Software for Vulnerability

Dynamic analysis is very useful to detect and prevent vulnerabilities in software. When a program is in execution, if violation occur they can be detected and possibly dealt with. This section will look at a few tools that were developed to protect an operating system from attack at runtime.

4.1 Software Security using Software Dynamic Translation [1]

In this paper they present a security API, which is built upon STRATA, a utility used for software dynamic translation. When a program is run inside the STRATA environment it can be instrumented and its execution controlled. What this means is that if vulnerability is exploited within a program it can be trapped before it becomes a problem, or even avoided altogether.

They extend STRATA with four functions:

```
void init_syscall();
watch_syscall(unsigned num, void *callback);
void strata_policy_begin(unsigned num);
void strata_policy_end(unsigned num);
```

A sample usage will look like this (some code is omitted):

```
strata_policy begin(SYS_open);
makepath_absolute(absfilename, path, 1024);
if(strcmp(absfilename, "/etc/passwd") == 0)
  { strata_fatal("Naughty, naughty!");
  }
fd = syscall("SYS_open, path, oflag");

strata_policy_end(SYS_open);
```

In this example we can see a file is being opened, however STRATA can catch the file, and check to see if it is opening /etc/passwd (a file which should never be opened in this application). If a violation occurs the program exits and "`Naughty, naughty!`" is displayed.

If this was implemented correctly then it should virtually eliminate security problems found in code. However, there is one major weakness. It depends on the programmers to implement these calls in their code. The majority of security vulnerabilities come from sloppy programming to begin with. By adding complexity, more problems may be added. Also there is overhead by running applications within STRATA, although dynamic optimizations could be applied to ease overhead, and possibly speed up, execution time.

4.2 **A Secure Environment for Untrusted Helper Applications** [2]

The goal of this work is to create a harmless workspace for untrusted applications. Again, they wish to confine the program from use of the operating system, which would limit the amount of possible damage. Also they use the "keep it simple" slogan for their work, acknowledging that complexity adds vulnerability. Also they wish to add versatility and configurability, so a user and specify any type of system calls to be monitored.

They implement a tool named Janus to do just this. There are two parts to Janus, the framework and the dynamic modules. The framework reads in a configuration file, which specifies certain rules to be followed within the program, and the dynamic modules implement the security policy.

Janus runs the applications inside this secure environment, catching any peculiarities when running the application. Because Janus is interpreting and monitoring the program, optimizations are also run, to make up for the performance loss.

They succeed in their goals. Only minimal overhead is needed to implement Janus, and creates a secure environment for several tests. One limitation is that it only works well for programs that do not require many privileges.

4.3 **Transparent Run-Time Defense Against Stack Smashing Attacks** [4]

Two methods are discussed here to eliminate buffer overflows in software. First, intercept system calls that are known to cause problems, and second, verify the contents of the stack before execution.

The first method is inspired by the C programming languages insecurity. Two problems they cite with C are, that it does not automatically perform bound checking on arrays/pointers, and also many of the functions in the standard C libraries are unsafe, such as `strcpy`, `strcat`, `gets`, `fscanf`, `scanf`, etc... What they attempt to do is intercept these calls, and execute them. If a buffer overflow occurs, the overflow will not leave the current stack frame. This is implemented by a dynamically loadable library, *libsafe*. Libsafe dynamically estimates safe upper-bound on each buffer being processed. The goal is to allocate space without overwriting the return address of the function, thus eliminating the key ability of a buffer overflow attack, being able to jump out of a function and execute arbitrary code.

The second method is implemented by using another dynamically loadable library, *libverify*, to verify critical elements in the stack before execution. Libverify implements a method similar to StackGuard, where the return address of a function is validated before a return is made. The difference is that, where StackGuard used a precompiled canary value to verify the return address, libverify dynamically inserts the code at the start of each process, and uses the actual return address of the function.

The results are as expected. Libsafe is fast, however works only with calls which were previously known to be vulnerable. Libverify is slower but can secure a greater set of applications. In tests libverify ran up to 15% slower, which in some cases could be unacceptable.

4.4 **Understanding and Managing Polymorphic Viruses** [6]

This paper is an overview of the Symantec's Striker System. The problem of polymorphic virii, is that the payload is encrypted, and the mutation engine constantly changes (see section 2). Assuming that the mutation engine is built well enough that the next variation of it cannot be detected (which is yet to happen), how could a polymorphic virus be detected?

One method of detecting polymorphic virii is by use of the principle of generic decryption, which states these assumptions:

- The body of a polymorphic virus is encrypted to avoid detection
- A polymorphic virus must decrypt before it can execute normally
- Once an infected program begins to execute, a polymorphic virus must immediately usurp control of the computer to decrypt the virus body, then yield control of the computer to the decrypted virus

Each file being scanned is placed inside a virtual computer (VC), and then executed. If there is no virus inside the file, then the VC ends the execution of the program, and continues to the next. However, if a virus is contained within the file the VC can easily find it and extract it from the file. The problem of using generic decryption is speed.

To combat the speed problem heuristics may be used. Heuristics that extend generic decryption are those, which look for inconsistent behavior within programs. One such example is that a regular program may perform calculations and then use these later on. A virus may perform similar calculations, for obfuscation, however may throw the values away. The problem with heuristics is that they must be continuously updated and researched.

The Striker System uses a method similar to generic decryption, however instead of using heuristics, it works with profiles of virii. When it scans a new file, the Striker System attempts to rule out as many virii as it can. The file will be continued to run inside the VC until all possible virii are eliminated as potential choices, and then moves to the next file.

4.5 **Data mining for Detection of New Malicious Executables** [8]

Currently the standard approach to finding unknown malicious executables has been through heuristics that miss often and are costly to use. Data mining techniques can also be applied to finding malicious executables. The goal of this work is to gain better performance over the current heuristics.

Their dataset, consisting of malicious and non-malicious executables, was split into two sets, the *training set* and the *test set*. The training set was used to build up a rule base to be tested against the test set, as is the standard data mining approach. The data from each executable was gotten in three ways, LibBFD, GNU Strings, and HexDump.

LibBFD was used to extract information from each executable, such as file size, the names of DLL's used, and the names of function calls within each DLL. This information gives insight into what types of operations a program will be performing.

GNU Strings was used to extract information from the executables. Surprisingly enough, certain distinctive strings were found across many malicious programs.

Lastly, HexDump was used to gain understanding as to what sequences of instructions are used in the program. This was the most informative phase in the learning process, because many malicious executables execute similar instructions that other programs do not.

After testing this technique found twice as many malicious executables as current methods. The results showed that about 216 out of 270 new malicious executables could be caught, which keeps up with the rate that they are produced every month, (about 240-300). Current methods can only catch about 87-109 of these new executables.

The drawback is practicality. A huge training set is necessary to produce such results. For most users, this is usually not an option.

5.0 Conclusions and Future Work

Much work has been done to secure software, however there is still much work that can be done. The reasoning behind this statement is that security breaches happen on a daily basis. If one of these methods was guaranteed to work and was easy to use, then the number of breaches would drop.

Much of this work focuses on buffer overflows. Buffer overflows are the most common type of attack, however once that problem is dealt with then another form of attack would become the most common. A universal tool needs to be created that prevents not only buffer overflows, but also a wide variety of attacks.

Speed and overhead is also a problem for many of these tools. In the corporate world, where web surfers hit sites at the rate of 1000's of times a minute, milliseconds of execution time are important. Speed over security is a common attitude. New techniques need to be developed which do not increase overhead tremendously, however still give high levels of security. Optimizations may be the key to help existing work to gain speed and performance.

Ease of use and simplicity are other key issues. No one wants to insert code/comments into pre-existing systems to make them secure. The time and effort are too great. Also more complex solutions mean a greater price tag. In addition, complexity adds vulnerability. It would be quite difficult to exploit a one line "Hello World" program, however once the complexity increases, such as by storing "Hello World" into a pointer, asking the user for input, etc.., there is potential for vulnerability.

Work currently being done on STRATA in relation to breakpoints may be an option to create a more secure environment. By using STRATA to encompass a program and break at certain points, such as system calls, the points where a program attempts to interact with the operating system can be detected. If vulnerability is found at these places then action can be taken without harm to the operating system. This differs from the STRATA security API, because it does not require extra code to be added into the software.

The way functions are called can also be researched. The problem with vulnerable functions is that it can be jumped out of and arbitrary code executed. If it was possible to specify with each function call that jumps can only occur at these specific memory addresses to other specified addresses, perhaps arbitrary jumps can be avoided. Another modification of this is for each program to have specific areas in memory where jumps can occur. This would limit the room an exploit had to work with and possibly stop it in its tracks.

The future will yield countless vulnerabilities and new types of attacks. Software is not getting more secure, however security can be gained by adding simple tools built around vulnerable software.

6.0 References

[1] Kevin Scott and Jack W. Davidson. Software Security using Software Dynamic Translation. University of Virginia CS Technical Report CS-2001-29

[2] Ian Goldberg, David Wagner. Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In Proceedings of the 1996 USENIX Security Symposium, 1996.

[3] David Larochelle and Devid Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, 2001.

[4] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defecse Against Stack Smashing Attacks. In proceedings of the 200 USENIX Annual Technical Conference (USENIX-00), pages 251-262, Berkeley, CA, June 18-23 200. USENIX Ass.

[5] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Symposium (SECURITY-98), pages 63-78, Berkeley, January 26-29 1998. Usenix Association.

[6] Carey Nachenberg. Understanding and Managing Polymorphic Visuses. The Symantec Enterprise Papers Volume XXX. 1996.

[7] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. IEEE Symposium on Security and Privacy. 2001.

[8] Matthew G. Schultz, Eleazar Eskin, and Erez Zadok. Data Mining Methods for Detection of New Malicious Executables. 2001.

[9] Aleph One. Smashing The Stack For Fun And Profit. Phrack Volume Seven, Issue Forty-Nine. 1996.

[10] SecurityFocus Vulns Stats. http://online.securityfocus.com/vulns/stats.shtml. 2001.

[11] SecurityFocus home vulns archive. http://online.securityfocus.com/bid. 2002.

[12] David Litchfield. Windows 2000 Format String Vulnerabilities. http://packetstormsecurity.nl/papers/Win2000/win32format.doc. 2001.

[13] Mark Nielsen. Chrooting All Services in Linux. http://www.gnujobs.com/Articles/23/chroot.html. 2001.