
EXPL

OITING SOFTWARE:

The Achilles' Heel of CyberDefense

By Gary McGraw and Greg Hoglund

Teaching and writing about attacks has always been controversial. However, if we are to make progress on the software security problem (arguably the most critical computer security problem of our time) we must understand how real software attacks happen. Pervasive misconceptions persist regarding the capabilities of software exploits. Many people don't realize how dangerous a software attacker can be. Nor do they realize that few of the classic network security technologies available today do much to stop them. Perhaps this is because software seems like magic to most people, or perhaps it's the misinformation and mismarketing perpetuated by unscrupulous (or possibly only clueless) security vendors. This article provides a brief introduction to the idea of exploiting software.

On Information Warfare

The second oldest profession is war. But even a profession as ancient as war has its modern cyber-instantiation. Information Warfare (IW) is essential to every nation and corporation that intends to thrive (and survive) in the modern world [Denning, 1998]. Even if a nation is not building IW capability, it can be assured that its enemies are, and that the nation will be at a distinct disadvantage in future wars.

Intelligence gathering is crucial to war. Since IW is clearly all about information, it is also deeply intertwined with intelligence gathering. Classic espionage has four major purposes:

1. National defense (and national security)
2. Assistance in a military operation
3. Expansion of political influence and market share
4. Increase in economic power

An effective spy has always been someone who can gather and perhaps even control vast amounts of sensitive information. In this age of highly interconnected computation, this is especially true. If sensitive information can be obtained over networks, a spy need not be physically exposed. Less exposure means less chance of being caught or otherwise compromised. It also means that an intelligence gathering capability costs far less than has traditionally been the case.

Since war is intimately tied to the economy, electronic warfare is in many cases concerned with the electronic representation of money. For the most part, modern money is a cloud of electrons that happen to be in the right place at the right time. Trillions of electronic dollars flow into and out of nations every day. Controlling

the global networks means controlling the global economy. That turns out to be a major goal of information warfare.

Digital Tradecraft and Software

Some aspects of information warfare are best thought of as digital tradecraft. Modern espionage is carried out using software. In an information system driven attack, an existing software weakness is exploited to gain access to information, or a backdoor is inserted into the software before it's deployed. Existing software weaknesses range from configuration problems to programming bugs and design flaws. In some cases, the attacker can simply request information from target software and get results. In other cases, subversive code must be introduced into the system. Some people have tried to classify subversive code into categories such as: logic bomb, spyware, Trojan Horse, etc. The fact is that subversive code can perform almost any nefarious activity. Thus any attempt at categorization is most often wasted exercise if you are concerned only with results. In some cases, broad classification helps users and analysts differentiate attacks, which may aid in understanding. At the highest level, subversive code performs any combination of the following activities:

1. **data collection**
   a. packet sniffing
   b. keystroke monitoring
   c. database siphoning
2. **stealth**
   a. hiding data (stashing log files, etc)
   b. hiding processes
   c. hiding users of a system
   d. hiding a digital "dead drop"
3. **covert communication**
   a. allowing remote access without detection
   b. transferring sensitive data out of the system
   c. covert channels and steganography
4. **command and control**
   a. allowing remote control of a software system
   b. sabotage (variation of command and control)
   c. denying system control (denial of service)

New books such as *Exploiting Software, The Shellcoder's Handbook*, and *How to Break Software Security* are all focused on the technical details of exploiting software in order to construct and introduce subversive code. The skills and techniques of software exploit have been employed by a small but growing community of people for almost twenty years. Many techniques were developed independently by small, disparate groups.

Only recently have software exploit techniques been combined into a single art. The coming together of disparate approaches is largely an historical accident. Many of the techniques for reverse engineering were developed as an off-shoot of the software-cracking movement that started in Europe. Techniques for writing subversive code are similar to techniques for cracking software protection (e.g., patching, etc.), so naturally the virus movement shares similar roots and core ideas. It was not uncommon in the '80s to find virus code and software cracks on the same bulletin-board systems (BBS's). Hacking network security, on the other hand, evolved out of the community of UNIX administrators. Many people familiar with classical network hacking think mostly of stealing passwords and building software trapdoors, for the most part ignoring subversive code. In the early '90s the two disciplines started to merge and the first remote shell exploits began to be distributed over the Internet.

Today, there are many books on computer security but few of them explain the offensive aspect from a technical programming perspective. All of the books on "hacking" (including the popular book Hacking Exposed) are compendiums of hacker scripts and existing exploits focused on network security issues. They do nothing to train the practitioner to find new software exploits. This is too bad, mostly because the people charged with writing secure systems have little idea what they are really up against. If we continue to defend only against the poorly-armed script kiddie, our defenses are not likely to hold up well against more the sophisticated attacks happening in the wild today.

The Trinity of Trouble: Making Software Security Difficult

Why is making software behave so hard? Three factors work together to make software risk management a major challenge today. We call these factors the trinity of trouble. They are:

- Complexity
- Extensibility
- Connectivity

Complexity

Modern software is complicated, and trends suggest that it will become even more complicated in the near future. For example, in 1983 Microsoft Word had only 27,000 lines of code, but according to Nathan Myhrvold, by 1995 it was up to 2 million! Software Engineers have spent years trying to figure out how to measure software. Entire books devoted to software metrics exist, our favorite one by Zuse weighing in at over 800 pages. Yet only one metric seems to correlate well with number of flaws: lines of code (LOC). In fact, LOC has become known in some hard core Software Engineering circles as the only reasonable metric.

The number of bugs per thousands lines of code will vary from system to system. Estimates are anywhere between 5 to 50 bugs per KLOC (thousand lines of code). Even a system that has undergone rigorous quality assurance (QA) testing will still contain bugs — around 5 bugs per KLOC. A software system that is only feature tested, like most commercial software, will have many more bugs — around 50 per KLOC. Most software products fall into the latter category. Many software vendors mistakenly believe they perform rigorous QA testing when in fact their methods are very superficial. A rigorous QA methodology goes well beyond unit testing and includes fault injection and failure analysis.

To give you an idea of how much software lives within complex machinery, consider the following:

LINES OF CODE	SYSTEM
400,000	Solaris 7
17 million	Netscape
40 million	Space Station
10 million	Space Shuttle
7 million	Boeing 777
35 million	NT5
1.5 million	Linux
Under 5 million	Windows 95
40 million	Windows XP

As we mention above, systems like these tend to have bug rates that vary between 5 and 50 per KLOC.

One demonstration of the increase in complexity over the years is to consider the number of LOC in various Microsoft operating systems. Figure 1 shows how the Microsoft Windows operating system has grown since its inception in 1990 as Windows 3.1 (3 million lines) to its current form as Windows XP in 2002 (40 million lines). One simple but unfortunate fact holds true for software: more lines, more bugs. If this fact continues to hold, XP is certainly not destined to be bug free! The obvious question to consider given our purposes is: how many such problems will result in security issues? And how are bugs and other weaknesses turned into exploits?

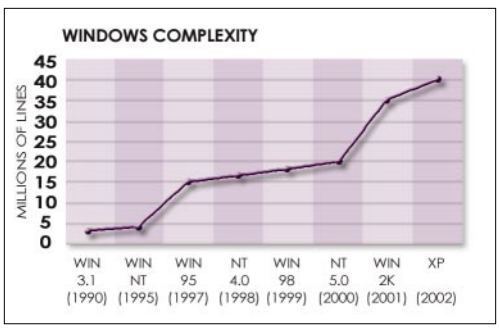


Figure 1: *Windows complexity as measured by Lines of Code (LOC). Increased complexity leads to more bugs and flaws.*

A desktop system running Windows XP and associated applications depends on the proper functioning of the kernel as well as the applications to ensure that an attacker cannot corrupt the system. However, XP itself consists of approximately 40 million lines of code, and applications are becoming equally, if not more, complex. When systems become this large, bugs cannot be avoided.

Exacerbating this problem is the widespread use of low-level programming languages such as C or C++ that do not protect against simple kinds of attacks such as buffer overflows [Hoglund and McGraw, 2004]. In addition to providing more avenues for attack through bugs and other design flaws, complex systems make it easier to hide or mask malicious code. In theory, we could analyze and prove that a small program was free of security problems, but this task is impossible for even the simplest desktop systems today, much less the enterprise-wide systems used by businesses or governments.

Extensibility

Modern systems built around virtual machines that preserve type safety and carry out runtime security access checks — in this way allowing untrusted mobile code to be executed — are extensible systems. Two prime examples are Java and .NET. An extensible host accepts updates or extensions, sometimes referred to as mobile code, so that the system's functionality can be evolved in an incremental fashion. For example, a Java VM will instantiate a class in a namespace and potentially allow other classes to interact with it.

Most modern operating systems support extensibility through dynamically loadable device drivers and modules. Today's applications, such as word processors, e-mail clients, spreadsheets, and Web browsers, support extensibility through scripting, controls, components, dynamically loadable libraries, and applets. But none of this is really new. In fact, if you think about it, software is really an extensibility vector for general purpose computers. Software programs define the behavior of a computer, and extend it in interesting and novel ways.

Unfortunately, the very nature of modern extensible systems makes security harder. For one thing, it is hard to prevent malicious code from slipping in as an unwanted extension, meaning the features designed to add extensibility to a system (such as Java's class loading mechanism) must be designed with security in mind. Furthermore, analyzing the security of an extensible system is much harder than analyzing a complete system that can't be changed. How can you take a look at code that has yet to arrive? Better yet, how can you even begin to anticipate every kind of mobile code that may arrive? These and other security issues surrounding mobile code are discussed at length in Securing Java

[McGraw and Felten, 1999].

Mobile code has a dark side that goes beyond the risks inherent in its design for extensibility. In some sense, viruses and worms are kinds of mobile code. That's why the addition of executable e-mail attachments and virtual machines that run code embedded on websites is a security nightmare. Classic vectors of the past, including the "sneakernet" and the infected executable swapped over modems, have been replaced by e-mail and web content. Mobile code based weapons are being used by the modern hacker underground. Attack viruses and attack worms don't simply propagate; they install backdoors, monitor systems, and compromise machines for later use in nefarious purposes.

... analyzing the security of an extensible system is much harder than analyzing a complete system that can't be changed. How can you take a look at code that has yet to arrive? Better yet, how can you even begin to anticipate every kind of mobile code that may arrive?

Viruses became very popular in the early 1990's and were mostly spread through infected executable files shuffled around on disks. A worm is a special kind of virus that spreads over networks and does not rely on file infection. Worms are a very dangerous twist on the classic virus and are especially important given our modern reliance on networks. Worm activity became widespread in the late '90s, although many dangerous worms were neither well publicized nor well understood. Since the early days, large advances have been made in worm technology. Worms allow an attacker to carpet-bomb a network in an unbridled exploration that attempts to exploit a given vulnerability as widely as possible. This amplifies the overall effect of an attack and achieves results that could never be obtained by manually hacking one machine at a time. Because of the successes of worm technology in the late '90s, most if not all global 1000 companies have been infected with backdoors. Rumors abound in the underground regarding the so-called Fortune 500 List – a list of currently working backdoors the Fortune 500 company networks.

One of the first stealthy malicious worms to infect the global network and be widely used as a hacking tool was written by a very secretive group in the hacker underground calling itself "ADM," short for "Association De Malfaiteurs." The worm, called "ADM w0rm" , exploits a buffer overflow vulnerability in domain-name servers (DNS). Once infected, the victim machine begins scanning for other vulnerable servers. Tens of thousands of machines were infected with this worm, but little mention of the worm ever made the press. Some of ADM's original victims remain infected to this day. Alarmingly, the DNS vulnerability used by the worm only scratched the surface. The worm itself was designed to allow other exploit techniques to be added to its arsenal easily. The worm itself was, in fact, an extensible system. We can only guess at how many versions of this worm are currently in use on the Internet today.

In 2001, a famous network worm called Code Red made headlines by infecting hundreds of thousands of servers. Code Red infects Microsoft IIS webservers by exploiting a very simple and unfortunately pervasive software problem. As is usually the case with a successful and highly publicized attack, several variations of this worm have been seen in the wild. Code Red infects a server and then begins scanning for additional targets. The original version of Code Red has a tendency to scan other machines that are in proximity to the infected network. This limits the speed with which standard Code Red spreads.

Promptly after its network debut, an improved version of Code Red was released which fixed this problem and added an optimized scanning algorithm to the mix. This further increases the speed at which Code Red infects systems. The success of the Code Red worm rests on a very simple software flaw that has been widely exploited for over twenty years. The fact that a large number of Windows-based machines share the flaw certainly helped Code Red spread as quickly as it did.

Similar outbreaks of worms include Blaster and Slammer.

The growing connectivity of computers through the Internet has increased both the number of attack vectors (avenues for attack) and the ease with which an attack can be made. Connections range from home PCs to systems that control critical infrastructures (such as the power grid). The high degree of connectivity makes it possible for small failures to propagate and cause massive outages. History has proven this with telephone network outages and power system grid failures as discussed on COMP.RISKS and in the book Computer Related Risks.

Connectivity

Because access through a network does not require human intervention, launching automated attacks is relatively easy. Automated attacks change the threat landscape. Consider very early forms of hacking. In 1975, of you wanted to make free phone calls you needed a "blue box". The "blue box" could be purchased on a college campus, but you needed to find a dealer. They also cost money. This meant that only a few people had blue boxes and the threat propagated slowly. Contrast that to today; if a vulnerability is uncovered that

allows attackers to steal pay-per-view television, the information can be posted on a web site and a million people can download the exploit in a matter of hours, deeply impacting profits immediately.

New protocols and delivery mediums are under constant development. The upshot of this is more code that hasn't been well tested. New devices are under development that can connect your refrigerator to the manufacturer. Your cellular phone has an embedded OS complete with file system. Figure 2 shows a particularly advanced new phone. Imagine what will happen when a virus infects the cellular phone network?

Figure 2: *A complex mobile phone offered by Nokia. As phones gain functionality such as e-mail and web browsing, they become more susceptible to software exploit.*

Highly connected networks are especially vulnerable to service outages in the face of network worms. One paradox of networking is that high connectivity is a classic mechanism for increasing availability and reliability; but path diversity also leads to direct increase in worm survivability.

Finally, the most important aspect of the global network is economic. Every economy on earth is connected to every other. Billions of dollars flow through this network every second; trillions of dollars every day. The SWIFT network alone, which connects 7000 international financial companies, moves trillions of dollars every day. Within this interconnected system, huge numbers of software systems connect to one another and communicate in a massive stream of numbers. Nations and multi-national corporations are dependant on this modern information fabric.

A glitch in this system can produce instant catastrophe, destabilizing entire economies in seconds. A cascading failure could well bring the entire virtual world to a grinding halt. Arguably, one target of the despicable act of terrorism on September 11, 2001, was to disrupt the world financial system. This is a modern risk that we must face up to.

The public may never know how many software attacks are leveraged against the financial system every day. Banks are very good about keeping this information secret. Given that network-enabled computers have been confiscated from many convicted criminals and known terrorists, it would not be surprising to learn that criminal and terrorist activity includes attacks on financial networks.

The Upshot

Taken together, the trinity of trouble has a deep impact on software security. The three trends of growing system complexity, built-in extensibility, and ubiquitous networking make the software security problem more urgent than ever. Unfortunately for the good guys, the trinity of trouble has a tendency to make exploiting software much easier!

Bugs and Flaws Are Distinct

A bug is a software problem. Bugs may exist in code and never be executed. Though the term bug is applied quite generally by many software practitioners, we'll reserve use of the term in this article to encompass fairly simple implementation problems that impact security. For example, misusing strcpy() in C and C++ in such a way that a buffer overflow condition exists is a bug [Viega and McGraw, 2001]. For us, bugs are implementation level problems that can be easily "squashed". Bugs can exist only in code. Designs do not have bugs. Code scanners such as Fortify's Dev tool are great at finding bugs.

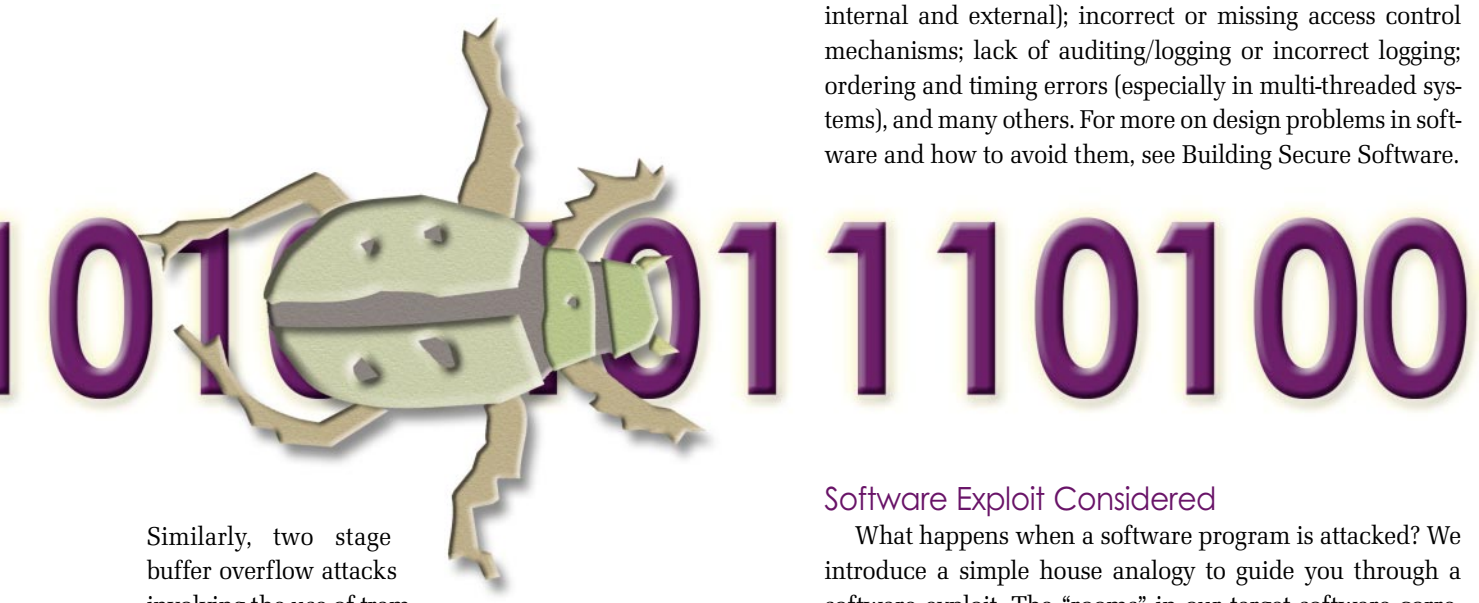
A flaw is also a software problem, but a flaw is a problem at a deeper level. Flaws are often much more subtle than simply an off-by-one error in array reference or use of a dangerous system call. A flaw is instantiated in software code, but is also present (or absent!) at the design level. For example a number of classic flaws exist in error handling and recovery systems that fail in an insecure fashion. Another example is exposure to cross site scripting attacks through poor design. Flaws may exist in software and never be exploited.

Bugs and flaws are vulnerabilities. A vulnerability is a problem that can be exploited by an attacker. There are many kinds of vulnerability. Computer security researchers have created taxonomies of vulnerabilities, though general agreement on categorization has not emerged.

Security vulnerabilities in software systems range from local implementation errors (e.g., use of the gets() function call in C/C++), through interprocedural interface errors (e.g., a race condition between an access control check and a file operation), to much higher design-level mistakes (e.g., error handling and recovery systems that fail in an insecure fashion or object sharing systems that mistakenly include transitive trust issues).

Attackers generally don't care whether a vulnerability is due to a flaw or a bug, though bugs tend to be easier to exploit. Some vulnerabilities can be directly and completely exploited, others only provide a toehold for a more complex attack.

Because attacks are becoming more sophisticated, the notion of what kind of vulnerabilities actually matter is constantly changing. Timing attacks are now common, whereas only a few years ago they were considered exotic.



Similarly, two stage buffer overflow attacks involving the use of trampolines were once the domain of software scientists, but are now used in 0day exploits [Hoglund and McGraw, 2004].

Design-level vulnerabilities are the hardest category of defect to deal with. Unfortunately, ascertaining whether or not a program has design-level vulnerabilities requires great expertise. This makes finding design-level flaws not only hard to do, but particularly hard to automate. Design-level problems appear to be prevalent and are at the very least a critical category of security risk in code. Microsoft reports that around 50% of the problems uncovered during the "security push" of 2002 were design-level problems.

Consider an error handling and recovery system. Failure recovery is an essential aspect of security engineering. But it's also complicated, requiring interaction between failure models, redundant designs, and defense against denial of service attacks. In an object-oriented program, understanding whether or not an error handling and recovery system is secure involves ascertaining a property or properties spread throughout a multitude of classes which are themselves spread throughout the design. Error detection code is usually present in each object and method, and error handling code is usually separate and distinct from the detection code. Sometimes exceptions propagate up to the system level and are handled by the machine running the code (e.g., Java 2 VM exception handling). This makes determining whether a given error handling and recovery design is secure quite difficult. This problem is exacerbated in transaction-based systems commonly used in commercial e-commerce solutions where functionality is distributed among many different components running on several servers.

Other examples of design-level problems include: object sharing and trust issues; unprotected data channels (both internal and external); incorrect or missing access control mechanisms; lack of auditing/logging or incorrect logging; ordering and timing errors (especially in multi-threaded systems), and many others. For more on design problems in software and how to avoid them, see Building Secure Software.

Software Exploit Considered

What happens when a software program is attacked? We introduce a simple house analogy to guide you through a software exploit. The "rooms" in our target software correspond to blocks of code in the software that perform some function. The job at hand is to understand enough about the rooms to wander through the house at will.

Each block of code (room) serves a unique purpose to the program. Some code blocks read data from the network. If these blocks are rooms in a house and the attacker is standing outside the door on the porch, then networking code can be thought of as the foyer. Such network code will be the first code to examine and respond to a remote attacker's input. In most cases, the network code merely accepts input and packages it into a data stream. This stream is then passed deeper into the house, to more complex code segments that parse the data. So the (network code) foyer is connected by internal doorways to adjacent, more complex rooms. In the foyer, not much of interest to our attack can be accomplished, but directly connected to the foyer is a kitchen with many appliances. We like the kitchen, because the kitchen can, for example, open files and query databases. The attacker's goal is to find a path through the foyer into the kitchen.

The Attacker's Viewpoint

An attack starts with breaking rules and undermining assumptions. One of the key assumptions to test is the "implicit trust" assumption. Attackers will always break any rule relating to when, where and what is "allowed" to be submitted as input. For the same reasons that software blueprints are rarely made, software is only rarely subjected to extensive "stress testing", especially stress testing that involves purposefully presenting malicious input. The upshot is that users are, for reasons of inherent laziness, trusted by default. A implicitly trusted user is trusted to supply correctly formed data that plays by the rules and is

thus also implicitly "trusted".

To make this clearer, we'll restate what's going on. The base assumption we'll work against is that trusted users will not supply "malformed" or "malicious" data! One particular form of this trust involves client software. If client software is written to send only certain commands, implicit assumptions are often made by the architects that a reasonable user will only use the client software to access the server. The issue that goes unnoticed is that attackers usually write software. Clever attackers can write their own client software, or hack up an existing client. An attacker can (and will) craft custom client software capable of delivering malformed input on purpose, and at just the right time. This is how the fabric of trust unravels.

Terminology for the Upcoming Battle

Though novelty is always welcome, techniques for exploiting software tend to be few in number and fairly specific. This means that applying common techniques often results in the discovery of new software exploits. A particular exploit usually amounts to the extension of a standard attack pattern to a new target. Classic bugs and other flaws can thus be leveraged to hide data, escape detection, insert commands, exploit databases, and inject viruses. Clearly, the best way to learn to exploit software is to familiarize yourself with standard techniques and attack patterns and come to see how they are instantiated in particular exploits.

An attack pattern is blueprint for exploiting a software vulnerability. As such, an attack pattern describes several critical features of the vulnerability and arms an attacker with the knowledge required to exploit the target system. A list of attack patterns can be found in the sidebar.

Our use of the term pattern is after Gamma [Gamma et al, 1995]. An attack pattern is like a pattern in sewing, a blueprint for creating a kind of attack. Everyone's favorite example, buffer overflow attacks, follow several different standard patterns. Patterns allow for a fair amount of variation on a theme. They can take into account many dimensions, including: timing, resources required, techniques, etc.

Exploit, Attack, and Attacker

An exploit is an instance of attack pattern created to compromise a particular piece of target software. Exploits are typically codified into easy to use tools or programs. Keeping exploits as stand-alone programs is usually a reasonable idea as they can in this way be easily organized and accessed.

An attack is the act of carrying out an exploit. This term can also be used loosely to mean exploit. Attacks are events that expose a software system's inherent logical errors and invalid states.

Finally, an attacker is the person who uses an exploit to carry out an attack. Attackers are not necessarily malicious, though there is no avoiding the connotations of the word. Notice that in our use of the term, script kiddies and those who are not capable of creating attack patterns and exploits themselves still qualify as attackers! It is the attacker who poses a direct threat to the target system. Every attack has an intent which is guided by a human. Without an attacker, an attack pattern is simply a plan. The attacker puts the plan into action. Each attack can be described relative to vulnerabilities in the target system. The attacker may restrict or enable an attack, depending on skill level and knowledge. Skilled attackers do a better job of instantiating an attack pattern than unskilled attackers do.

Conclusion

Exploiting software is an art and a challenge. First you have to figure out what a piece of code is doing, often by observing it run. Sometimes you can crash it and look at the pieces. Sometimes you can send it crazy input and watch it spin off into oblivion. Sometimes you can disassemble it, decompile it, put it in a jar and poke it with experimental probes. Sometimes (especially if you are a white hat) you can look at the design and spot architectural problems.

If we are to defend our systems against sophisticated software exploits, we must familiarize ourselves with the tactics of the enemy. Nothing short of this will work. CDM

References

• Dorothy E. Denning (1998) Information Warfare & Security, Addison-Wesley, New York.
• Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995) Design Patterns, Addison-Wesley, New York.
• Greg Hoglund and Gary McGraw (2004) Exploiting Software, Addison-Wesley, New York.
• Stuart McClure, Joel Scambray and George Kurtz (1999) Hacking Exposed: Network Security Secrets and Solutions, Osborne, New York.
• Gary McGraw and Ed Felten (1998) Securing Java: Getting down to business with mobile code, John Wiley and Sons, New York.
• Peter Neumann (1995) Computer Related Risks, Addison-Wesley (ACM Press), New York.
• Ken Thompson (1984) Reflections on Trusting Trust, Communications of the ACM, 27(8).
• John Viega and Gary McGraw (2001) Building Secure Software: How to avoid security problems the right way, Addison-Wesley Professional Computing Series, New York.
• James Whittaker and Herbert Thompson (2003) How to Break Software Security, Addison-Wesley, New York.
• Horst Zuse (1991) Software Complexity: Measures and Methods (Programming Complex Systems, No. 4), Walter de Gruyter, Inc., Berlin.

Portions of this article adapted by permission from Exploiting Software: How to Break Code by Greg Hoglund and Gary McGraw (Addison-Wesley, 2004).