Malware and Buffer Overflows

Types of Malware

Malware is a general term for software that does something undesirable on purpose. Most malware relies on human mistakes or on software vulnerabilities to get into a system. Once it's there, it usually has some type of payload that does something bad. Let's look at some of the different kinds of malware.

Viruses The key property of a virus is that it is self-replicating. When a virus gets onto a system, it looks for ways to copy itself to other things, often other files on the system. Many viruses attach a copy of themselves to executable files, so when that file is run, the virus's code is also run. Some viruses attach themselves to the boot sector of a computer, which is the code that is run when a computer first starts up. Others attach themselves to the boot sectors of disks and USB drives.

Macro viruses Many types of software come with scripting languages that allow you to do things that are not possible in the regular user interface of the software. Scripts written for this purpose are sometimes called *macros*. The most common type is probably the macro capability built into MS Office products. They use Visual Basic. Used properly, they can allow you to do some pretty awesome spreadsheet calculations and PowerPoint tricks. However, these macros often have access to many system resources. Some macros can run immediately when the document is opened. You might download a MS Word document from the internet, run it, and if a virus has been written in the macro's scripting language, then your computer has now been infected. This is why when you open downloaded Word documents, Word will often put up a warning message.

Worms A *worm* is similar to a virus in that it tries to spread itself. Viruses, like human viruses, need a host to infect. They are usually chunks of code that attach themselves to other files. A worm stands on its own without a host to attach to. Worms often try to propagate to other machines via a network.

Rootkits Rootkits are a type of malware that infects the operating system itself. They are usually broken into two types: *user-mode rootkits* and *kernel-mode rootkits*. The latter are the more dangerous as they infect the deepest part of the operating system and run with privileges even higher than that of the root user. Because they get so far into the operating system, they can perform all sorts of tricks to avoid being detected or deleted. For instance, they can block themselves from appearing in the task manager. Or when someone tries to delete a file associated with the rootkit, because the rootkit is running as part of the OS, it can make sure the deletion doesn't take place. Sometimes the only thing you can do to get rid of a rootkit is to reinstall the OS, though you may be able to locate and delete the rootkit files if you boot up using a different OS.

Trojans A *trojan*, like the mythological trojan horse, is malware that is disguised as something else. For instance, a program that claims to be a Minesweeper game but actually deletes all the files on your computer is a trojan.

Logic bombs A *logic bomb* is code that is set to run when triggered by a particular event. An example would be a software engineer who adds some code to a system that is set to delete important files if the engineer's name is ever removed from the database of employees. Another example would be a program that only executes its payload if it recognizes that it is in a certain country on a specific date.

Backdoors A *backdoor* isn't necessarily a program or a piece of malware, but it can have similar effects. A backdoor is something that gives access via other means than the usual way. For instance, if a router manufacturer sets a very specific username and password that gives access to the router's settings, and the manufacturer doesn't tell anyone about it, that is a backdoor. Another example is if a company allows SSH access to their servers to someone with username "Smith" at port 44503. When malware infects a system, it can set up a backdoor to the system by opening up a particular port. There is currently a heated debate between governments and security researchers about putting government-mandated backdoors into cryptography to allow governments to decrypt things.

Spyware *Spyware* is a type of malware that spies on a user's activities and sends the info to an attacker. Keyloggers are one example. They silently record all the keys you type and periodically send them to an attacker.

Ransomware *Ransomware* is a type of malware that encrypts your data and forces you to pay the attacker money in order to get your data decrypted.

More about malware

Antivirus programs usually work by looking for signatures, usually blocks of code that are part of a virus. Virus creators get around this by modifying the code to change the signature. There are point-and-click programs out there for virus creation that allow people to create new viruses even without knowing how to code. They're not creating new exploits, but rather are creating viruses using preprogrammed exploits that have new signatures.

Some viruses can modify themselves to avoid detection. They can encrypt or obfuscate their code, and change parts of it on the fly. Others update themselves via the internet. Viruses that can change shape like this are called *polymorphic*.

Most malware relies on some type of vulnerability in a system. If that vulnerability has not previously been known to the company producing the system, then it's called a *zero-day* since the company has known about it for zero days and hence hasn't had any time to fix it. There's an active group of people coming up with zero-days. Some people disclose them to the companies and then publish them after patches are available. Often companies will reward people with bug bounties for this, though some companies take a less enlightened approach. Other people sell their zero-days on the black market.

One of the goals of a lot of modern malware is to make a device part of a *botnet*. A botnet is a network of compromised machines that all obey a bot master. The master issues instructions and the bots all follow those instructions. Bots can be computers, routers, and internet of things devices. Many of these devices get infected due to obvious default passwords and due to not being patched with the most recent security updates. It's also easy to get infected by clicking on a bad link or downloading something from the wrong place.

A look at some famous pieces of malware

Here we will give a brief overview of some of the more well-known pieces of malware throughout the last 30+ years.

Morris worm (1988) This is the first really well-known worm. It was not intentionally malicious, but it did affect a large percentage of the internet at the time due to a bug that caused it to infect machines multiple times. It relied on a few different attack vectors. One was a buffer overflow in the Unix finger program. Another was the fact that the sendmail program, if it was running in debug mode, would automatically run executable files sent in the subject line. Many machines were running it in debug mode. Once the worm infected a machine, it opening connections to other computers with the remote shell (rsh) program by guessing passwords. This was another way it coud spread.

ILOVEYOU worm (2000) Versions of Microsoft Outlook around 2000 had a bug where if the file name of an attachment had two extensions, it wouldn't show the second one. The ILOVEYOU worm started out in emails that had an attachment called LOVELETTERFORYOU.txt.vbs. The second extension, the true one, is for a Visual

Basic script. But Outlook only showed the file as LOVELETTERFORYOU.txt, making it look like an innocent text file. Once run, the script would send copies of the worm to people in the victim's address book, making it look like they were the ones sending the "love letter". Besides this, the worm would overwrite various files with copies of itself.

Code red worm (2001) This ia worm that relied on a buffer overflow in a specific version of Microsoft web server software. The worm would try sending a specific HTTP request to random IP addresses. If it found one running the vulnerable software, the HTTP request would trigger the buffer overflow. The worm would then vandalize the page and sometimes try to do a denial of service at a few targets, including the White House. The worm itself only lived in RAM and not on the hard drive. When the server was restarted, the worm would be removed, but the machine would soon be reinfected from the internet.

Blaster worm (2003) Microsoft had released a security update. A group reverse-engineered the update to figure out what the bug was, and they developed it into an exploit. That exploit was a buffer overflow in remote procedure calls (RPCs). Because people are often slow to update their systems, the worm that used the exploit was able to spread widely. Once it got onto a network, it could spread very quickly to the rest of the network due to RPCs being allowed within the network.

One interesting development was the Welchia good worm. It used the same exploit to spread, but its payload was to actually patch the system so it couldn't be exploited by the Blaster worm. This brings up interesting ethical questions.

Sony rootkit (2005 As a way to prevent people from illegally copying CDs, Sony included a rootkit that would install itself when a CD was played on a computer. Attackers found ways to use the rootkit to help their own malware. Sony's attempts to patch the problem caused even more problems.

Conficker (2008) People think this worm may have been released by a government organization as a test. Like the Blaster worm, it relied on a buffer overflow in Windows remote procedure calls. The worm used a number of techniques including disabling Windows updates and blocking DNS queries for antivirus sites. It would infect USB drives, and it updated itself from the internet via randomly generated domain names.

Stuxnet (2010) This widely believed to be developed by U.S. and Israeli agencies to target Iran's nuclear reactors. The worm spread far and wide, but it had no payload unless it detected that it was running on certain nuclear reactor control systems. It was able to cross air gaps (i.e., reach computers not connected to the internet) by infecting USB drives.

Wannacry ransomware (2017) This also exploits a buffer overflow in Windows remote procedure calls. Patches for the exploit had been released about a month before the worm began spreading, but many people didn't update their systems. The worm had a kill switch built into it. If it wasn't able to reach a certain domain, it would shut itself off. This was a protective measure that the worm used to detect if it was being run in a virtual machine, which is what researchers use to safely study worms. A security researcher discovered the domain name, registered it, and set the DNS to sinkhole it (so it would go to 0.0.0.0). This stopped the worm from spreading. Its creators tried using new domains, but each time they would be registered and sinkholed.

Buffer overflows

As we saw above, most of the worms relied on buffer overflows. What exactly is a buffer overflow? In computer science, a *buffer* is an area of memory, usually an array or a string. An *overflow* is where you try to store more stuff in the buffer than it has room for. That extra stuff can spill over into adjacent memory locations, potentially overwriting other variables, or worse.

The C and C++ languages are particularly vulnerable to buffer overflows. Other languages, like Python and Java are not so vulnerable. In Python, if you declare a list `L = [0,1,2]` and try to write to `L[10]`, you will get an index out of bounds exception, and nothing will be written. But C and C++ will gladly try to write to that index with no warnings or exceptions at all. Here is an example of some code that suffers from a buffer overflow.

```
#include <stdio.h>
#include <string.h>

int main() {
    char buffer1[10];
    char buffer2[10];
    strcpy(buffer1, "123456789");
    strcpy(buffer2, "abcdefghijklmnopqrst");
    printf("buffer1: %s\n", buffer1);
    printf("buffer2: %s\n", buffer2);
    return 0;
}
```

We have allocated space for 10 characters in `buffer2`, but we try to store 20 characters in it. Some of those characters will overflow into the memory location for `buffer1`, and instead of being `"123456789"`, it ends up being `"qrst"` The buffers are stored on a stack, which why `buffer2` overflows into `buffer1` and not the other way around.

Programs use a stack to store information needed for function calls. Each function gets a chunk of it called a *stack frame*. That stack frame is where it stores all its local variables and some other things, the most important of which is the *return address*. This indicates where in the program's code to return to when the function is done. Buffer overflows usually try to overflow a buffer far enough to overwrite the return address with the address of some code the attacker wants to run. That code could be another function in the program, it could be a library function, or it could be code the attacker has placed in the buffer. Here is an example a program being exploited by a buffer overflow.

```
./vulnerable_program $(python -c "print('\x90'*40
 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
 \x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'
  + 'A'*40
  + '\x20\xd3\xff\xff')")
```

The person doing this is using Python to send some data to the program. Everything in the print statement is what is being put into the buffer. Most of it, all the `\x##` stuff, is raw machine language instructions. The part starting with `\x31` on the second line is *shellcode*, whose purpose is to open up a command shell. This will give the attacker access to the system to run commands. These commands will run with the same level of permission as the vulnerable program itself. So if it's running as root, then the attacker will have root privileges on the system and can basically do anything. Some descriptions of vulnerabilities refer to this "arbitrary code execution". After the shellcode, the program puts in a bunch of A's. The purpose of this is to overflow the buffer to right where the return address is. The last part, on the last line, is the address of the buffer itself in memory. This points the return address to the buffer so that the shellcode the attacker put into memory will be run.

Here is another example. This is what the Code Red worm would send to random IP addresses. It's an HTTP request with the name of the resource being requested designed to cause a buffer overflow. We see a certain number of N's being used to overflow the buffer right to where the return address is. What follows that is the raw machine code to be run.

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

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

Buffer overflows are one the most common and most serious vulnerabilities in modern software. A lot of system software is written in C and C++ because it is fast and gives you access to the system itself in a way that high-level languages like Java and Python can't. But in C and C++, it's very easy to accidentally create an opportunity for buffer overflows.