Co-funded by the
Erasmus+ Programme
of the European Union

ISSES – Information Security Services
Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant
N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

MALWARE II

COMPUTER SECURITY

Lecture 13

ISSES 2017-2020
Novi Sad, Serbia

In this lecture…

- In this lecture we'll take a look at a number of historically significant pieces of malware.
- We'll be focusing on worms, most of all, and on those with significant impact or technical novelty.
- We won't analyze code closely, but will instead focus primarily on the structure of the worm, how it operates, and what its advent meant for the advancement of malware and methods used to defend against it.
- The lecture will conclude with a sort of overall conclusion to the entire lecture series which will attempt to summarize the chief insight this whole course was meant to engender into a simple-to-remember slogan.

Information Security Services Education in Serbia (ISSES)

13.1 The Great Worms: Morris and Slammer

Knocking down the Internet

- We mentioned it in our history lecture way at the start of the course: The Morris Worm was so effective that it managed to, effectively, shut down the Internet.
- The outage lasted for several days.
- This was possible in 1988. It's difficult to imagine such success would be possible today.
- How did the Morris worm work?
- It'd run on a machine and scan for vulnerable hosts it has access to.
- Then it would use a selection of three exploits to jump over to a new host. Once it was on a new host it'd start work anew, looking for vulnerable hosts to infect.
- It had no malicious payload: it just expanded without limit.

Exploit #1: Sendmail

- Sendmail is a mail transfer agent (an MTA). This is an app which relays e-mail. If the e-mail is for a local user it sends it to their local inbox, and if it is for a remote one, provided it is configured in relay mode, it sends it on to the target host.
- At the time if Sendmail was in debug mode (a lot of instances were, as Sendmail had a *notoriously* opaque configuration syntax) you could send the host the name of an executable as the recipient of the message provided you first sent a ‘debug’ instruction.
- Sendmail would then *run the executable* and feed it the message body as input.
- This is… not a security *flaw* so much as a giant open invitation to be hacked.

Why the exploit?

- The question arises, how come such a blatant, obvious, horrible exploit was left on sufficient machines?
- The answer is best summarized in this direct quote from Eugene H. Spafford's "The Internet Worm Program: An Analysis" published as a Purdue Technical Report CSD-TR-823, see: https://spaf.cerias.purdue.edu/tech-reps/823.pdf

"Yet, despite its importance and wide-spread use, most system administrators know little about how it works. Stories are often related about how system administrators will attempt to write new device drivers or otherwise modify the kernel of the OS, yet they will not willingly attempt to modify sendmail or its configuration files."

Exploit #2: Finger

- The somewhat obscurely named ‘finger’ program was a utility to query information about other users of a computer system. It was meant to display simple directory information: Office, phone-number, and so on.
- What finger (which ran as a deamon on port 79 back in the day) had wrong with it is very simple: it used *gets.*
- Gets does no bound checking, and is, therefore, an invitation for a buffer overflow attack.
- This is what the Worm used: it sent shellcode, stole access, and sent itself across to the shell it just spawned.
- The shell would then run the worm and hide all traces of it running.

Why the exploit?

- We *still* have problems with buffer overflows.
- C/C++ is just inherently memory unsafe in its operation.
- To make matters worse this was way, way before any of the protections we were discussing until recently were implemented.
- No stack canaries, no noexec, no protection whatsoever.
- Breaking in was a simple exercise of finding an overflow and sending a specially-crafted string, and gaining access.
- It's intriguing that even at the time everyone already knew gets was unsafe and that you can break in through overflows and people *kept* using gets and only broke the habit, mostly, recently.
- Programmer inertia is a dangerous thing.

Exploit #3 RSH

- RSH (remote shell) is something we mentioned during our hardening lecture as something to disable.
- At this stage RSH was still a going concern and this program used it to attempt to establish shell access by guessing passwords.
- It'd make guessing more efficient by using RSH configuration files to guess which systems were 'equivalent' i.e. shared a user base.
- This was, by today's standards, a very primitive attack, consisting of a dictionary of common words alongside variations on a user's name.
- Despite this, it was found that this attack was successful in 30% of the cases: this is what not having a strict password policy does.

Why the exploit?

- Because security is overhead.
- To a user, every security practice is something they *can't* do or some chore they *can't* avoid.
- Having an empty password or a password they can easily remember is something a user likes.
- Having a password with X words and Y capital letters and Z numbers and special symbols is a horrible imposition.
- This is not as bad today as it was in 1988: users are, generally, more security conscious, but it only takes one careless user in a multi-user system to effect at least a minor compromise.
- This is why implementing a stern password security policy is so important.

Where the worm failed

- The worm, we now know, was designed as an intellectual exercise by then undergrad student Robert Tappan Morris.
- It has a secondary potential use of being a tangible ‘I Told You So’ for the security flaws Morris discovered.
- It *wasn’t* meant to bring down the whole Internet.
- What went wrong?
- *The worm had no reinfection check limiter.*
- Or, rather, the limiter was incorrectly coded: this means that the worm often reinfected machines it had infected before, causing more and more load on the machines and more and more load on the network.
- Accidentally, Morris invented the distributed denial of service attack.

The Slammer Worm

- Let's skip ahead in history and look at the Slammer Worm.
- 1988 was still a time when the Internet was young. Slammer comes from 2003 when the commercialization of the Internet was progressing apace, but hadn't reached anything like its current state.
- Slammer infected Microsoft SQL Server 2000 machines.
- MS SQL Server was (and still is) a piece of relational database management software.
- It receives queries, written in a standard, SQL language, and dispatches responses.
- MS SQL Server 2000 had an additional service: a UDP online service which allowed rapid search among the names of individual databases.

Guess who's back?

- The UDP service had a buffer overflow bug.
- The rules of the protocol were:
  - The first byte must be 0x04
  - The remainder of the datagram payload must be
    - At most 16 bytes
    - Terminated by a NUL character.
  - This remainder should be the database being sought by whoever was running the directory search.
- Whoever wrote the code actually allocated as much as 128 bytes as an input buffer.
- Naturally, this is not enough.

Infection vector

- The vector is very simple.
- The attack was just randomly destructive with no actual payload.
- As a result, there's no cause for shellcode.
- Instead, the attacking code sent a 0x04 byte, and then a very long string of 0x01 bytes to fill out the buffer.
- Then it sent the overwrite portion of the code which would be used as an address to jump back to.
- This code would then, in an infinite loop, generate an IP address randomly, and send a similarly crafted UDP packet to it too.
- That's it: it doesn't do anything, just spreads rapidly and consumes resources.

Aftermath

- This couldn’t quite cause the Internet to fail.
- However, it did noticeably slow down its general speed.
- Why? Because there was no waiting period and UDP is asynchronous.
- This means that, in its attempt to spread, it also started *hammering* every other system on the internet.
- In other words, due to a different sort of lack of rate limiting, it turned itself into a yet another DDOS attack.
- This, incidentally, happened despite the patch for the vulnerability it was exploiting having been released six months previous.

Information Security Services Education in Serbia (ISSES)

13.2 CONFICKER

A jump in sophistication

- We’re jumping ahead to 2009.
- The internet as you know it is taking shape.
- This is not an amateur effort: serious sophistication went into this attack.
- It was speculated, that the Conficker Worm was let loose by an entity backed by a government as a field test of a cyber-warfare attack.
- This theory comes from an observed similarity between the way Conficker and Stuxnet (see 13.3) worms operate, especially how they infect their hosts.
- Officially, however, the creation of the worm is attributed to an Ukranian organized criminal group since it would not infect systems with Ukranian keyboard layouts and Ukranian addressese.

What Conficker did

- So far our worms were without payload: all they wanted was to spread.
- Conficker was more malign: its payload surrendered complete control of the system to the attacker.
- This was only occasionally full manual control by the worm's authors.
- More often the worm would perform a series of automated control steps:

1. Disable Windows Update
2. Make the machine unable to carry out a DNS lookup of hosts associated with antivirus software manufacturers.
3. Lock out certain user accounts.

Conficker versions

- Conficker came in a number of versions usually coded Conficker.A, Conficker.B, Conficker.C, Conficker.D, and Conficker.E.
- A, B, C, and D were straight updates and, somewhat uniquely for worms thus far, older versions will actively update to newer versions.
- This means as computers get better fighting the infection, so does the infection, fetching updates from online update servers.
- Conficker E was meant to coexist with Conficker D and deploy an additional malware payload and then remove itself.

Principal infection vector

- The systems were ultimately infected through a vulnerability of the svchost process.
- Svchost, also known as a ‘generic host process’ is a process which runs services on a Windows machine.
- Services in Windows aren’t executables but DLLs which can’t be run on their own but are sort of *attached* to a svchost process.
- Svchost figures out what to run and how by consulting the services subsection of the registry and loading in and executing the DLLs specified.
- It is a weakness in the way svchost works and how it handles remote procedure calls that lets the worm gain an initial foothold.

Vector #1: Svchost.exe

- If a machine has a version of svchost.exe which has not been patched, this is the method used to attack.
- It attacks port 445 which is owned by the SMB protocol Windows networks use to share resources: files, printers and so on.
- A part of this protocol is remote procedure calls: the ability to execute code remotely in well-defined and, in theory, secure ways.
- This lets multiple windows hosts function more like one large system.
- It’s a vulnerability here that the worm exploits: if the request path is crafted in a very particular way there is an overflow during the path canonization step that allows unauthenticated remote code execution.

Vector #1: Svchost.exe

- The problem was in the NetpwPathCanonicalize() function hosted in netapi32.dll.
- Canonicalization is a technique which converts a path into ‘canonical’ form by resolving paths like \a\b\..\c\.\d into \a\c\d.
- This function could be served a too-large path crafted just so which causes a standard stack overflow.
- Using this it’s possible to seize control over the return address and use a sort of nested return-to-libc-style attack by altering the stack and return address to invoke the URLDownloadToFile() functions and the LoadLibrary() functions

Vector #1: Svchost.exe

- The sequence of steps, therefore, when infecting through this vector was:
  - Gain access through port 445
  - Load a malware DLL from a given HTTP URL.
  - Execute a new instance of svchost process and use it to host the downloaded DLL file
  - Configure the registry so that this new service, randomly named, runs at boot.
  - Continue propagating using one of the three vectors.
- This was most commonly named the MS08-067 propagation mode after the vulnerability it eexploited.

Vector #2: NetBIOS

- Once the worm was on a system it would exploit the network of trust built around such a system to try to propagate further, possibly even to systems which were not vulnerable to its principal propagation vector.
- One method was to seek out accessible Windows Shares especially those locally mapped as logical volumes (the equivalent to mounting a NFS share in Linux) and drop the copy of the malware there.
- Sometimes this required passwords, and therefore the bot came equipped with a list of 240 commonly used passwords it would try in order in a very small-scale dictionary attack.
- This was *annoyingly* successful.

Here we are again

- It's been (in our history) 21 years since the Morris worm and *still* a small dictionary is enough to launch an attack.
- Twenty-one years and still no improvement in overall password security.
- The lesson to be learned here is that sometimes it is the obvious things that are an attack vector.
- *Of course* your users will set a proper password… right?
- You can patch MS08-067.
- You can't patch users who do not follow security guidelines.
- Vector #2 was known as the NetBIOS propagation mode.

Vector #3: USB

- The malware would place a copy of itself as the autorun.inf file on any USB drives plugged in.
- This would run the worm as soon as you plugged the infected USB into a computer.
- This is a way to propagate further and is a way to infect even airgapped computers.
- Of course, users aren't meant to plug *anything* into an airgapped computer without checking it first, carefully, but on the other hand, users are meant to use sensible passwords a dictionary with 240 entries won't be able to crack and yet…
- This vector was known as the USB propagation mode, for obvious reasons.

Updates and command-and-control

- Nobody is entirely sure how the owners of the worm managed to communicate with it.
- The updating mechanism was somewhat better known, however.
- It downloaded fresh versions (which were encrypted and digitally signed which, honestly, *all software downloaded online should be*) from random subdomains of several major TLDs.
- These subdomains were random, but were synchronized among all Conficker instances by using the current date as a pseudorandom number generator seed.
- This meant all instances tried to download from the same place.

Novelty of Conficker

- There's nothing novel about exploiting a buffer overflow vulnerability in a worm.
- That's how the very first worm spread, after all.
- What makes Conficker worthy of study and consideration aren't is specific vulnerabilities but how it operated as a complete package.
- Worms we've studied thus far have been relatively simple, mostly interested in propagating and doing so in relatively simple ways.
- Conficker was outstanding in its aggressiveness, combining multiple attack vectors, the capability to update itself and keep evolving past the ability security software to contain.

Novelty of Conficker

- In a sense, Conficker operated as its own miniaturized version of an advanced persistent threat.
- It would attack using multiple vectors, making sure to compromise a system any way it could.
- It would hide on the system very effectively.
- It would stay updated and change its payload and behavior.
- It would sabotage any attempt to control it.
- It's this all-in-one multi-stage approach of Conficker that would be important to future development of malware.
- It's no longer a malicious piece of software as it is an entire hostile system with multiple modules that communicate.
- In this wise, the structure of malware has almost followed the development of software engineering practice.

Information Security Services Education in Serbia (ISSES)

13.3 STUXNET

A new sort of target

- Stuxnet is unique among things we talked about in its target
- It attacks neither Unix or Windows general computing hosts but instead targets SCADA systems.
- SCADA stands for Supervisory Control and Data Acquisition.
- Less blandly put SCADA is *the* standard for controlling large industrial systems with automation.
- A SCADA installation allows a small team of operators to exert complete control of an entire massive factory, power plant, or some other industrial facility from a single control point using computerized systems to both acquire data and control desired outputs.
- SCADAs are the *ne plus ultra* of modern automation.

SCADA is a big deal

- There is a separate subject which covers this sort of security of industrial processes in great, exhaustive detail, but for this let us simply grasp the *scale* of the problem of this sort of attack.
- SCADA is used in power plants, it's used in large factories, in critical water-treatment infrastructure, in air treatment, especially in large systems with complicated requirements such as hospitals.
- If it is complicated, industrial, and involves a lot of components interacting, esp. those distributed geographically chances are it uses SCADA.
- An attack focused on such systems is one with nearly incalculable capacity to cause harm.

Cyber-warfare?

- Nobody *knows* this, and there is no actual proof, but it is a popular piece of conjecture that Stuxnet was launched very deliberately as an attack on the nuclear processing capacity of certain countries.
- Again: there is no *proof* of this but the worm has had this effect, and it seems to have an oddly specific target to its payload.
- It's meant to infect personal computers on a local network and then seek out very specifically Siemens computers used for SCADA.
- It would then spoof sensor data (to obscure its actions from any vigilant human operators) and launch a very specific attack meant to destroy connected nuclear-enrichment hardware.

A physical vulnerability

- The reason this attack was so effective is that an important step to manufacturing nuclear materials is *enrichment.*
- For this step ^{235}U and ^{238}U have to be separated out. These are isotopes and have no chemical differences that can be used to differentiate between them. The only thing that can be used, practically, is mass.
- Therefore Uranium Hexafluoride is fed into giant, massive cylinders which spin at something like more than 50 000 rotations per minute held aloft on magnetic bearings.
- The difference in centripetal force because of mass allows the separation of a ^{235}U-rich and ^{235}U-poor streams and the mechanism is cascaded until the desired purity is achieved.
- If this is not managed *extremely carefully,* however, this system is prone to fail catastrophically.

Stuxnet payload

- Stuxnet used the vulnerability of this system to effect the physical destruction of the centrifuges by varying their speed suddenly, either to speeds that are too large for safe operation or to speeds that are too low, causing sudden friction between components that aren’t meant to touch.
- The wear caused by this increases the likelihood of fatal accidents which, given the speeds at which this sort of device operates, is likely to involve explosive equipment failure.
- This is done by compromising the Programmable Logic Controller (PLC) component of a SCADA system, specifically, the Step7 software by Siemens running on computers controlling PLCs.

How does Stuxnet propagate?

- Like Conficker before it Stuxnet uses multiple attack vectors.
- It uses the good old MS08-067 vulnerability that we have covered already and that Conficker used as well.
- But it also uses tow additional vulnerabilities as propagation vectors:
  - MS 10-046 (CVE-2010-2568) Windows shortcut vulnerability
  - MS 10-061 (CVE-2010-2729) Print spooler vulnerability
- It uses any of these methods it can to reach new hosts, phone home (using a pre-set server) with infection data, and spread through a network until a computer with PLC access can be found where stage two of the worm can be deployed.

How does Stuxnet propagate?

- According to the detailed Symantec research report (now only accessible through archive at: https://web.archive.org/web/20190710235415/http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf) the system also propagates by:
  - Copying over shares like Conficker
  - Copying on USB with an autorun.inf file, like Conficker
  - Copies itself into Step 7 projects so that it executes whenever such a project is loaded.
  - Uses a WinCC vulnerability (to be discussed later) to infect systems that way.
  - Uses a local privilege escalation attack MS10-073 to ensure it has full system access.

MS 10-046

- You can defend against a malicious USB by disabling autorun.
- MS 10-046 allows us to work around this, by instead crafting a malicious shortcut (LNK file) which is so malformed that when Windows attempts to process it (so it knows what it is displaying) it instead executes arbitrary code.
- *Everything* the system processes counts as input, and any bit of input could potentially trigger overflows and hijack the system.
- This is particularly dangerous as the user does not need to do anything which the user might view as a positive, permissive action that they need to be careful about. Merely viewing the file is enough.

MS 10-061

- This is a close cousin to MS08-067.
- It's also a SMB RPC vulnerability (yes, another one) which sabotages the validation of user permissions and lets unauthenticated users 'print' by writing files to system locations.
- Carefully placing a file into the right system location can make it execute automatically, meaning that 'printing' the right file can make it execute, enabling RCE-with-extra-steps.
- The lesson here is that if there's any protocol that needs to be locked down, it's SMB.
- Backwards compatibility is a big reason why: if Microsoft was making it from scratch now it'd probably be a lot more secure.

Second stage

- Once it has infected hosts, it seeks out more hosts and so on until it detects a system with PLC access.
- It does so by searching the Windows system folder for a file S7OTBXDX.DLL whichi ndicates the presence of a Siemens Step7 installation. It then renames this file to hide it from the system and creates a new file named S7OTBXDX.DLL which is exactly like the original except with modification to the code of certain fundamental functions having to do with reading, writing, and in generall accessing blocks of code on the attached PLC.
- The system then launches stages three and four. Stage four is only launched on systems which have a very specific hardware configuration, indicating a highly targeted nature.

Stage three

- Stage three gets full control of the database of the WinCC SCADA control software.
- It does so through an exploit of CVE-2010-2772.
- What's CVE-2010-2772? A hardcoded password.
- That's it. There is one password that allows full access to the database and it's hardcoded in the software.
- This is a very common sort of fault: someone thinks that something isn't worth protecting and so trades security for convenience and it all works fine until people who look for these things for a living ferret it out.
- To write safe code, you *must* treat every piece of code as potentially targets of attack.

Stage four

- Stage four is a bit of a mystery as the full source code to the worm was never publicly released.
- What is known is that it intercepts requests to communicate with PLCs from a legitimate program and then injects its own payload into the PLC before calling the backup copy of the DLL it replaced to execute what the user wanted.
- It only deploys this highly specific payload (PLCs and the hardware they help control can vary wildly under certain circumstances) under very specific conditions making it highly targeted.
- The sources on this topic are scant, but some requirements have leaked into the public record through the work of Eric Chien of Symantec.

Stage four requirements

- S7-300 CPU
- CP-342-5 Profibus comm module
- The hardware controlled must be a frequency converter drive which varies the speed at which something driven by alternating current runs by varying the frequency.
- The hardware being controlled must be from specific vendors (one Finnish and one Iranian)
- The frequency must be between 807 Hz and 1210 Hz indicating an incredibly high speed of operation.
- Normal industrial systems don’t operate on these frequencies, and any drive that runs at higher than 600Hz is considered a regulated good for purposes of nuclear non-proliferation.

Stage four effects

- Stuxnet runs a loop of expected sensor output while, behind the scenes, and over a period of months, varying the output frequency for very short periods going from normal operation to 1410Hz (too fast), then 2Hz (too slow), then 1064Hz.
- Needless to say, this sort of rough treatment is precisely what you don’t want happening to a rapidly spinning tube full of heated radioactive gas.
- Given its highly targeted and involved nature, these effects must be the point of the worm. It was built to attack a very specific hardware configuration.

The aftermath

- What did Stuxnet bring?
- It's extremely sophisticated, but it doesn't do much more than Conficker did.
- It's more elaborate than Conficker and uses more vectors to spread and mount its attacks, certainly, but this is a question of degree, not of kind.
- What was new about Stuxnet, above all, was how *targeted* it is.
- In an essence this was, even more than Conficker, an advanced persistent threat in a single automated package.
- The authors didn't mind collateral damage (since it infected computers all over the world, a lot of them without any PLC hardware whatsoever) as long as it also got to hit its target.
- This even more than Conficker, is a fully automated, contained APT.

Information Security Services Education in Serbia (ISSES)

13.4 Conclusion

What have we learned?

- Our time together is nearly at an end – this is the last section of the last lecture of the course.
- What have we learned for all our time together?
- While it is true we've learned about a great number of vulnerabilities and great number of very malicious pieces of code, not to mention CVEs by the score, that's not what's truly important.
- Keeping information on security current is a never-ending task.
- As soon as one problem has been resolved, another pops up so any details you've learned will have to be refreshed and updated and added to for the entirety of your professional career.

What have we learned?

What I hope we've learned, most of all, is not just some facts and some attacks and some computer system features, but a certain *philosophy* of computer security that can help you direct further independent study and that can help you in your future carrer. We can encode these lessons in a somewhat glib pentad of Be's:

1. Be current.
2. Be humble.
3. Be meticulous.
4. Be creative.
5. Be conservative.

Be current

- Computer security, above all other fields, *does not sit still.*
- Certainly, computer science advances, but if you've learned to be a competent C++ programmer in 2004 or so, your skills are still in demand and you can make code that will still be very useful.
- If your computer security development has stopped in 2004, however, you are not only not in demand, your advice and security solutions would be *dangerous.*
- Whole new *classes* of attack have been discovered since and whole new types of defense.
- If you are going to deal with security it is imperative, therefore, to keep current on all the vulnerabilities being exploited and all the developments being made.

Be humble

- The cost of overconfidence in security is prohibitive and the number of possible attacks so vast that it is just about guaranteed that you will miss some avenue of attack.
- You need to simply take that into account and move on.
- This isn't about second-guessing or being frozen with indecision, this is about realizing that you have probably made mistakes and planning your system defense accordingly.
- This methodological humility is a good way to do engineering in general, but in security it is imperative.
- Methodologically humble software will continue to provide security features even when some part of security breaks because you've always assumed some of it must.

Be meticulous

- An important praxis when implementing security is being able to tolerate dullness.
- Sometimes it's not about brilliant, not about being cunning, not about being smart, but about patiently closing all the six thousand minor security defects, making sure not to miss any.
- Sometimes it is about double-checking or making enough of a pest of yourself that you get everyone to follow the security policy you've drafted.
- Your foes only need to be good once. You need to be good *all the time.*

Be creative

- The truly dangerous attack is the one that comes from an unexpected direction.
- Not just something that hasn't been patched but something you never thought to defend in the first place.
- This sort of out-of-the-box thinking is needed to be a truly successful attacker and is, therefore, just as needed in defense.
- You need to be able to make realistic predictions of directions attacks that haven't been invented yet might come from.
- This requires that you keep a flexible outlook and be creative.

Be conservative

- Programmers, legend says, are all haunted by a little demon known as the Creeping Feature Creature. This malign spirit creeps close and whispers "You know, your code would be so much better if it could *also* do *this.*" Over time, this inevitably leads to Emacs.
- Jokes aside, it is natural to want your system to do more.
- Often, your job as a security professional is to fight that urge and strip the system down, fighting in particular the urge to add things that aren't meant to be used immediately but are meant for some unspecified future use. Just-in-case features.
- These little-used features are the ones most likely to be un-upgraded or misconfigured or overlooked and those are prime attack sites.