My favorite huge number - script

Script

What do hackers, a multinational gaming company, and this xkcd comic have to do with cryptography? Watch this video where I explain this in the story about my favourite mega number.

<Title screen>

The Sony Playstation 3 was released in 2006 and just like with previous consoles it was an interesting target for hackers who wanted to run homebrew, pirate games or just tackle the challenge of breaking its protections. Over the years many different attacks have been performed against the Playstation 3 and in this video we will look at a particularly interesting one from a cryptography standpoint. In December 2010 the hacker group called fail0verflow gave a presentation at the Chaos Computer Congress in Germany where they describe a number of different vulnerabilities and exploits on the Playstation 3. As the grand finale of their presentation, they revealed that Sony had made a mistake in their implementation of the cryptographic algorithm used to create signatures in the console leading to not just breaking individual signatures but a full recovery of the private key allowing anyone to create signatures just as valid as the ones from Sony. They did not reveal any keys themselves during the presentation, however a few days later famous hacker George Hotz, known as geohot posted a set of keys on his website thanking fail0verflow for their work.

So, how does this flaw work and what does it mean in the context of the Playstation 3? To answer this we must first look at two things: the Playstation 3 chain of trust and some elliptic curve cryptography.

When the PS3 boots it does so in several stages. First the Secure Boot stage which is stored directly in hardware starts executing. It will read the Boot Loader from read only memory, verify its integrity, decrypt and execute it. The Boot Loader will in turn load the Level 0 loader, verify its integrity, decrypt and execute it. Finally, the Level 0 loader will then read the Meta Loader, verify its integrity, decrypt and execute it. The Meta Loader then goes on and performs similar operations, loading more pieces of code but this is as far as we are really concerned. As you can see, the security of the system relies on this chain where it is assumed that the first step is not tampered with and that each step then verifies the integrity of the next step before executing it. This means that it is not possible to replace for example the Meta Loader since the integrity check will fail and the boot process will be aborted.

To break this chain of trust we either need to modify the root of the chain which in this case is code etched directly into the hardware of the system or circumvent one of the integrity checks to allow us to modify one of the links in the chain without it being detected and this second option is exactly what Sony’s mistake enabled.

The integrity check is performed with what's called Elliptic Curve Cryptography, ECC for short, specifically, it uses the ECDSA algorithm which stands for Elliptic Curve Digital Signature Algorithm. It is one of several algorithms used to verify the integrity of a piece of data. Furthermore, this is an example of an asymmetric signature scheme which means that the author of some data can use a private key to create a digital signature and publish the data together with the signature. Anyone in possession of the other half of the key, the public key, is then able to verify this signature to guarantee the authenticity of the data (or detect that it has been tampered with). The nice thing about this is that someone who holds the public key can verify a signature but not create a new signature for that key.

To understand ECDSA we will first go through the basics of elliptic curves. In general, an elliptic curve is a set of points satisfying the equation "y^2+ay=x^3+bx^2+cxy+dx+e" however the ones we are interested in specifically, satisfies the simpler equation "y^2=x^3+ax+b" where x and y are coordinates and a and b are parameters of the curve. In these examples, let us pick a to be -5 and b to be 8 thus giving us the equation "y^2=x^3-5x+8". Let's draw this curve. We can now create a system where we can pick two points P, with coordinates x1 and y1, and Q, with coordinates x2 and y2, on this curve and add them together. This addition however does not work the same way you probably are used to where you just add up the x coordinates and the y coordinates separately. No, this is a different kind of addition which is a little bit more complicated.

First we draw a line through P and Q. This line will intersect the curve at a third point. From there we draw a new vertical line and follow it to where it crosses the curve again. This is the result of the addition. We call this point R. ~~The point where the first line crosses the curve can be called -R since it is reflected along the x axis. Note that in this context, that minus means that we negate only the y coordinate, not both coordinates as you might be used to.~~ For this to work completely we also have to introduce another special point that we call O or 0. This point can be thought of as lying infinitely far away from the curve. ~~The name comes from the fact that it behaves similar to 0 in regular addition. For example, if P or Q lies on the curve in such a way that the line drawn doesn't cross the curve again, the result is the 0 point. Adding the 0 point to another point P gives that point P as a result.~~ There are a few other cases to consider and I have put some links in the description to more thorough explanations including nice interactive tools. Of course doing this in a graphical way like this is a good visualization to get a feel for what's going on but when doing this in the computer we instead use a couple of different formulas that can be derived from this general idea.

This is all well and good and serves as a nice graphical introduction to elliptic curves, but this is not how they are used in cryptography. Notice how we assumed that the points we were talking about were all points with real valued coordinates which is why we get this nice smooth curve when we plot it. What if we restrict ourselves to just integers? Then the curve would look something like this instead. Now, what if we further restricted ourselves by taking the integers 0 up to 36 and instead did all the calculations modulo 37? The "curve" would then look like this instead. Now you might say: "hold up, that doesn't look like a curve at all." and I would agree but we still call this an elliptic curve and the calculations check out. Take this point (9, 10) for example, let's check that it is indeed on the curve which means that it should satisfy our equation y^2=x^3-5x+8. Let's start with the left hand side: 10^2=100

modulo 37 is 26. Now let's look at the right hand side: 9^3 is 729, -5*9 is 684, +8 is 692 modulo 37 is 26 so the left and right hand sides are indeed equal so the point is on the curve. This is true for all of the other points plotted here.

Now the cool thing with this is that the rules for addition described previously still work for these points even if they don't make as much sense visually. So far I've talked about addition, in which case we add two points together to get a third point as a result but what about multiplication? Is there an analogous operation in this context as well? The answer is yes. When multiplying regular integers we basically say, add the second number to itself as many times as this first number, for example 5x3 can be seen as "add three to itself, five times to get 15.". In a similar way we define multiplication of a point with an integer as "add this point to itself as many times as this number.", for example 3*P would mean, take the point P, add P to itself and then add P a third time to that result to get the answer.

Now armed with this knowledge we can finally get to the ECDSA algorithm. Previously I used some small numbers to easily illustrate how it all works but I will now use some of the numbers used in the actual Playstation 3 implementation. First we need to decide the parameters of the curve, this includes the numbers a and b and the modulo to use for our calculations. That modulo is called p. The following three huge numbers are used for this:

a = 1285827406856150272977634075244471302722353316650
b = 1745519616984928595610812197735018306278998308 90
p = 1285827406856150272977634075244471302722353316653

This also means that this curve contains an insane amount of points which is in a sense what adds to the security of this. We also need a point on this curve that is called the generator point or G for short. It can't be any point but has to satisfy some specific requirements which I won't go into in this video. In this case the point has the coordinates:

Gx = 122092236891752899337709727187901739803514721690 8
Gy = 663790648276954390506348183409422989997537118582
n = 1285827406856150272977634980925287584532553029089

That number n is the so called order of G. It means that if you add G to itself n times, i.e. multiply n and G, you get the 0 point.

These numbers are in theory completely public and there exist sets of these parameters which are standardized and given specific names such as secp256r1 and Curve25519. In the specific case of the Playstation 3 they are embedded in the console.

Next a private key is created by picking a random number between 1 and n. This private key is kept secret at Sony's headquarters and not published anywhere. Using the definition of multiplication from previous, the generator point is now multiplied with the private key D. The result is a point on the curve. This point is the public key which, again, is embedded in the console. In this case, the public key is:

d*G=Q
Qx = 1112286859332644437093155474859504663400020740370
Qy = 34864620313755224153757759633133640906292378421 7

Now this is where the security comes in. If you have G and the private key d it is easy to calculate the public key Q but if you have the public key Q and the point G there is no way to calculate what number G was multiplied with to generate Q. This is in contrast with regular multiplication where it would be easy to for example take 15, divide it by 3 and get 5 to see that 3 was added to itself 5 times to get 15.

The system only allows us to sign a number, specifically a number with the same number of bits as n. We want to sign a block of code, so how do we do this? First we run all the data through a cryptographic hash function and treat the output as a number. On the PS3, the SHA1 hash is used. Let's say that the program we want to sign is the string "Hello world!". We take the SHA1 hash of that string and treat it as a number.

SHA1("Hello world!") = d3486ae9136e7856bc42212385ea797094475802 =
1206212019512053528979580233526017047056064403458

This is the message we will sign. This number is called z. The next step is really important, keep it in mind, and it is to generate a random number between 1 and n. This number is named k. Let's do this.

k = 1065091044705817090501219694904053840157569858957

Now, we multiply the generator point G with this number:

k*G = (x1, y1)
x1 = 7332045592981043694436011776939828901956906813 67
y1 = 1163812768219003849359365017662955468850988162139

Then we discard y1 and calculate x1 modulo n and call this r:

x1 % n = r
r = 7332045592981043694436011776939828901956906813 67

If r happens to be 0 here we have to redo the previous two steps with a new random number. Finally we multiply the private key d with the random number r and add the message we want to sign z and multiply it all with the inverse of k mod n. This result is called s:

s = k^-1(z+r*d) (mod n)

s = 7588325803003600323560276176911920089861260415 6

Now we are done and the pair (r, s) is the signature. This is embedded together with the message to allow for checking its integrity. So how does that work? Let's say we want to verify that the signature (r, s) is valid for the message "Hello world!".

m = "Hello world!"
r = 733204559298104369443601177693982890195690681367
s = 758832580300360032356027617691192008986126041 56

We start by converting the message to a number by running it through a cryptographic hash, just like before:

z = SHA1("Hello world!") = d3486ae9136e7856bc42212385ea797094475802 = 1206212019512053528979580233526017047056064403458

Then we calculate u1 and u2 by taking z and r respectively and multiplying them with the inverse of s mod n.

u1 = z*s^-1 (mod n)
u1 = 1149460439399273418996929069965918001968428935616
u2 = r*s^-1 (mod n)
u2 = 2341375540854183541499058839990843848647229611 11

We then calculate a new point on the curve by multiplying u1 with G and u2 with the public key Q and adding those two points together:

u1 * g + u2 * Q = (x2, y2)
x2 = 733204559298104369443601177693982890195690681367
y2 = 1163812768219003849359365017662955468850988162139

We discard the y value of this point and take the x and compare it with the r value in the signature. If these two values are equal modulo n, the signature is valid, otherwise it is invalid

r = x2 (mod n) -> valid signature
r != x2 (mod n) -> invalid signature

x2 % n = 733204559298104369443601177693982890195690681367
r % n = 733204559298104369443601177693982890195690681367

The two numbers match and the signature is valid. Now, where did Sony go wrong in all of this? Remember in the signing phase when we needed to generate a random number and I said it was important. Let's say that number wasn't random at all but instead a fixed number that we used every time we signed something. What would that mean? Let's say we had signed another message to create a different signature (r', s'). Specifically, let's say we also have the signature for the message "Goodbye world!".

m = “Goodbye world!”
r’ = 733204559298104369443601177693982890195690681367
s’ = 607596597906974824949282696074916350534152407778

Notice how r’ is the same as before but s’ is different from s. We know that since r depends only on static variables, including the re-used number k, r and r’ will be the same

r = (k*G)_x (mod n)

so the two signatures are really (r, s) and (r, s’). Also recall how s was calculated:

s = k^-1*(z+r*d) (mod n)

If we take the difference between s and s’ we get

s - s’ = k^-1 * (z+r*d - z’ - r*d)

the two z are different since different messages were signed but the r*d terms cancel out leaving us with:

s - s’ = k^-1 * (z-z’)

using some algebra we can rearrange this into

(z - z’) * (s-s’)^-1 = k
k = 1065091044705817090501219694904053840157569858957

Notice how this is in fact the k that was used above. Now we can again look at how s is calculated:

s = k^-1 (z+r*d)

we can rearrange this into the following and plug in one of the signatures to calculate d:

(s*k-z)*r^-1 = d
d = 1128657407947805406850932552007336662296971380336

This means that by using signatures for two different messages which both used the same k value instead of a random one we were able to extract the private key. With this we can now sign any message we want.

This is exactly what happened in the Playstation 3. Sony used this very specific value for every signature instead of a random one each time. This was used to sign multiple things including the Meta Loader in the PS3 boot chain of trust. This means that by extracting the Meta Loader and their corresponding signatures from two different consoles it was possible to use this flaw to recover the secret key, create a new Meta Loader, sign it with that private

key and gain full control of any software run after that stage in the boot process, including running both pirated games and homebrew.

This is why that k number

k = 1065091044705817090501219694904053840157569858957

is my favourite mega number.

This video is part of a playlist under the hashtag #MegaFavNumbers where a group of maths youtubers and other people have created videos about their favourite number over one million. Check out the hashtag and the playlist for the other videos.

If you are interested in learning more about IT security and related topics consider subscribing to my channel and please leave a comment about what you thought about this video. Thank you for watching.

Notes

The key: “46 DC EA D3 17 FE 45 D8 09 23 EB 97 E4 95 64 10 D4 CD B2 C2” = “404555974007725459910684486621289147856453481154” = 4.04e47

“404 quattuordecillion 555 tredecillion 974 duodecillion 7 undecillion 725 decillion 459 nonillion 910 octillion 684 septillion 486 sextillion 621 quintillion 289 quadrillion 147 trillion 856 billion 453 million 481 thousand 154”

The random number “BA 90 55 91 68 61 B9 77 ED CB ED 92 00 50 92 F6 6C 7A 3D 8D”

- boot loaders
- chain of trust
    - PS3 chain (https://www.psdevwiki.com/ps3/Boot_Order#Chain_of_Trust)
    - Runtime Secure Boot > bootldr (Boot Loader) > lv0 (Level 0) > metldr (asecure_loader) > rest
    - breaking it
- cryptographically signed
- ECC basics
    - curve
    - addition
    - multiplication
    - curve over field
- ECDSA
    - curve, parameters
    - create private key
    - sign
    - verify

Sources

https://en.wikipedia.org/wiki/PlayStation_3_homebrew
https://yingtongli.me/blog/2019/01/28/crypto-failures.html
https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
https://www.youtube.com/watch?v=LP1t_pzxKyE
https://github.com/williamstein/480-ent-2014/blob/master/lectures/1780_27c3_console_hacking_2010.pdf
https://www.youtube.com/watch?v=siOXFGZj_z0
https://web.archive.org/web/20200220054602/http://psx-scene.com/forums/f6/geohot-here-your-ps3-root-key-now-hello-world-proof-74255/
https://web.archive.org/web/20110115203720/http://www.geohot.com/old_index.html
https://www.psdevwiki.com/ps3/Keys
https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/
https://andrea.corbellini.name/ecc/interactive/reals-add.html
https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/
https://www.psdevwiki.com/ps3/Boot_Order
https://m101.github.io/binholic/2011/11/16/ps3-hacking-part-1-exploitation.html
https://www.psdevwiki.com/ps3/images/e/ed/Ps3-cryptochain.png
https://imagequalitymatters.blogspot.com/2011/02/intrusion-prevention-ps3-re-secured.html
https://www.uni-regensburg.de/Fakultaeten/nat_Fak_I/friedl/papers/elliptic_2017.pdf
https://www.codecogs.com/latex/eqneditor.php?latex=D

“ERK/RIV is used to decrypt the encrypted SELF data”
erk (32 bytes): C0 CE FE 84 C2 27 F7 5B D0 7A 7E B8 46 50 9F 93 B2 38 E7 70 DA CB 9F F4 A3 88 F8 12 48 2B E2 1B
riv (16 bytes): 47 EE 74 54 E4 77 4C C9 B8 96 0C 7B 59 F4 C1 4D

pub: C2 D4 AA F3 19 35 50 19 AF 99 D4 4E 2B 58 CA 29 25 2C 89 12 3D 11 D6 21 8F 40 B1 38 CA B2 9B 71 01 F3 AE B7 2A 97 50 19
R: 80 6E 07 8F A1 52 97 90 CE 1A AE 02 BA DD 6F AA A6 AF 74 17
n: E1 3A 7E BC 3A CC EB 1C B5 6C C8 60 FC AB DB 6A 04 8C 55 E1
K: BA 90 55 91 68 61 B9 77 ED CB ED 92 00 50 92 F6 6C 7A 3D 8D
Da: C5 B2 BF A1 A4 13 DD 16 F2 6D 31 C0 F2 ED 47 20 DC FB 06 70

Timeline

11 Nov 2006 - PS3 released
Dec 2009 - Geohot announces attempts to hack PS3
22 Jan 2010 - Hypervisor escape
26 Jan 2010 - Release HV exploit
28 March 2010 - Sony announce removal of OtherOS

13 July 2010 - Geohot abandons hacking PS3
29 Dec 2010 - Fail0verflow presents ECC vuln at CCC, no keys released
2 Jan 2011 - Geohot releases private key on his website