Cryptographic Attacks: A Guide for the Perplexed

July 29, 2019

Research by: Ben Herzog

Introduction

When some people hear “Cryptography”, they think of their Wifi password, of the little green lock icon next to the address of their favorite website, and of the difficulty they’d face trying to snoop in other people’s email. Others may recall the litany of vulnerabilities of recent years that boasted a pithy acronym (DROWN, FREAK, POODLE…), a stylish logo and an urgent warning to update their web browser.

Cryptography is all these things, but it’s not *about* these things. It’s about the thin line between easy and difficult. Some things are easy to do, but difficult to undo: for instance, breaking an egg. Other things are easy to do, but difficult to do when a small, crucial piece is missing: for instance, unlocking your front door, with the crucial piece being the key. Cryptography studies these situations and the ways they can be used to obtain guarantees.

Over the years, the landscape of cryptographic attacks has become a kudzu plant of flashy logos, formula-dense whitepapers and a general gloomy feeling that everything is broken. But in truth, many of the attacks revolve around the same few unifying principles, and many of the interminable pages of formulas have a bottom line that doesn’t require a PhD to understand.

In this article series, we'll consider various types of cryptographic attacks, with a focus on the attacks' underlying principles. In broad strokes, and not exactly in that order, we'll cover:

- **Basic Attack Strategies** — Brute-force, frequency analysis, interpolation, downgrade & cross-protocol.
- **"Brand name" cryptographic vulnerabilities** — FREAK, CRIME, POODLE, DROWN, Logjam.
- **Advanced Attack Strategies** — Oracle (Vaudenay's Attack, Kelsey's Attack); meet-in-the-middle, birthday, statistical bias (differential cryptanalysis, integral cryptanalysis, etc.).
- **Side-channel attacks** and their close relatives, fault attacks.
- **Attacks on public-key cryptography** — Cube root, broadcast, related message, Coppersmith's attack, Pohlig-Hellman algorithm, number sieve, Wiener's attack, Bleichenbacher's attack.

This specific article covers the above material up until Kelsey's attack.

Basic Attack Strategies

The following attacks are simple, in the sense that they can be explained without many technical details, and without much of the substance being lost. We'll explain each type of attack in the simplest terms possible, without delving into complicated examples or advanced use cases.

Some of these attacks have mostly lost their relevance, and have not seen a successful mainstream application for many years. Others are perennials — they still routinely sneak up on unsuspecting cryptosystem designers in the 21st century. The modern era of cryptography can be considered to have begun with IBM's DES, the first cipher to withstand every attack on this list.

Simple Brute-Force Attack

An encryption scheme is made up of two parts: an encryption function, which takes a message (plaintext) in conjunction with a key, then produces an encrypted message (ciphertext); and a decryption function, which takes a ciphertext and a key, and produces a plaintext. Both encryption and decryption should be easy to compute, given the key – and difficult otherwise.

So, suppose we are looking at a ciphertext, and attempting to decrypt it without any additional information (this is called a "ciphertext only" attack). If we were somehow magically handed the correct encryption key, we would be able to easily verify that it is indeed the correct key: we'd decrypt the ciphertext using the proposed key, and then check whether the result is a reasonable message.

Note that we've made two implicit assumptions here. First, we assume that we know how to perform the decryption – that is, that we know how the cryptosystem works. This is a standard assumption when discussing cryptography. Hiding the cipher implementation details from attackers might seem to confer extra security, but once the attackers figure out these details, this extra security will be silently and irreversibly lost. That's Kerckhoffs' principle (https://en.wikipedia.org/wiki/Kerckhoffs%27s_principle): "The enemy knows the system."

Second, we assume that the correct key is the only key that will result in a reasonable decryption. That is also a reasonable assumption; it holds if the ciphertext is fairly long relatively to the key, and fairly legible. Generally, this is true in the real world, barring the use of a huge and impractical key (https://en.wikipedia.org/wiki/One-time_pad) or other shenanigans we had best leave out of this article (https://en.wikipedia.org/wiki/Deniable_encryption). (If the reader is unsatisfied with this hand-waving, please refer to theorem 3.8 here (https://www.eit.lth.se/fileadmin/eit/courses/edi051/lecture_notes/LN3.pdf)).

Given the above, a strategy emerges: iterate over every single key, and verify whether it is the correct key or not. This is called a brute-force attack, and it is guaranteed to work against all practical ciphers — eventually. For instance, a brute-force attack was powerful enough to defeat the shift cipher (https://www.codexpedia.com/cryptography/shift-ciphers/), an early cipher for which the key was a single letter out of the alphabet, which implies only twenty-something possible keys.

Unfortunately for cryptanalysts, a mitigation quickly presents itself: increasing the key size. As the size of the key grows, the number of possible keys increases exponentially. With modern key sizes, the naked brute-force attack is completely impractical. To understand what we mean by that, consider that the fastest known supercomputer as of mid-2019, IBM's summit (https://www.ibm.com/thought-leadership/summit-supercomputer/), has a peak speed on the order of 10^{17} operations per second, whereas a typical modern key length is 128 bits which translates to 2^{128} possible keys. Plugging in the numbers, if Summit were instructed to brute-force a modern key, the feat would require over 5,000 times the age of the universe.

Is the brute-force attack a historical curiosity? Far from it; it is a necessary ingredient in the cryptanalytic cookbook. Very few ciphers are so catastrophically weak that a clever attack completely breaks them, without requiring some elbow grease. Many successful breaks make use of a clever attack to weaken the targeted cipher, and then deliver a brute-force as the coup de grâce.

Frequency Analysis

Most texts are not gibberish. For instance, in English messages, you see a lot of the letter `e`, and a lot of the word `the`; in binary files, you see a lot of zero bytes, put there as filler between one chunk of information and the next. A frequency analysis is any attack that takes advantage of this fact.

The canonical example of a cipher vulnerable to this attack is the simple substitution cipher. In this cipher, the key is a table that, for each letter in the English alphabet, designates a letter to replace it with. For instance, `g` can be replaced with `h`, and `o` with `j`, so the word `go` becomes `hj`. This cipher resists a simple brute-force attack, as there are very many possible substitution tables (if you're interested in the math, the effective key length is about 88 bits — that's $\log_2(26!)$). But a frequency analysis typically makes short work of this cipher.

For example, consider the following ciphertext, encrypted with a simple substitution:

```
XDYLY ALY UGLY XDWNKE WN DYAJYN ANF YALXD DGLAXWG XDAN ALY FLYAUX GR WN OGQL ZDWBGEGZDO
```

As `Y` appears frequently and at the end many words, we can tentatively guess that its plaintext counterpart is the letter `e`:

```
XDeLe ALe UGLe XDWNKE WN DeAJeN ANF eALXD DGLAXWG XDAN ALe FLeAUX GR WN OGQL ZDWBGEGZDO
```

The pair `XD` repeats at the beginning of several words, and in particular the term `XDeLe` is strongly suggestive of a word such as `these` or `there`, and so we proceed:

```
theLe ALe UGLe thWNKE WN heAJeN ANF eALth DGLAtWG thAN ALe FLeAUt GR WN OGQL ZDWBGEGZDO
```

Next, we'll guess that `L` translates to `r`, `A` to `a`, and so on. Some trial-and-error will probably be involved, but compared to a full brute-force attack, this attack recovers the original plaintext in no time at all:

```
there are more things in heaven and earth horatio than are dreamt of in your philosophy
```

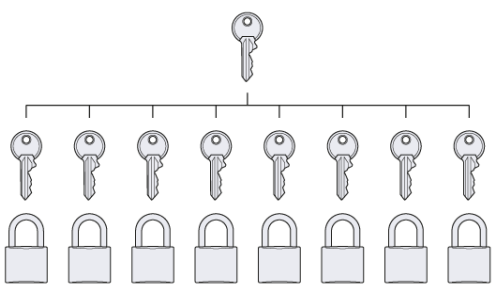
For some people, solving "cryptograms" such as the one above is a hobby.

The basic idea behind frequency analysis is more powerful than it appears at first sight, and is applicable to much more complex ciphers than the simple substitution above. Various cipher designs throughout history tried to counter the attack above via "polyalphabetic substitution" — that is, changing the letter substitution table mid-encryption in complex but predictable ways that depend on the key. These ciphers were all considered difficult to break in their time; and yet the humble frequency analysis attack eventually caught up with every single one.

The most ambitious polyalphabetic substitution cipher in history, and probably the most famous, was the Enigma cipher used in World War II. Its design was complex compared to those that came before it, but after much hand-wringing and toil, British cryptanalysts were able to break it using frequency analysis. Granted, they couldn't mount an elegant ciphertext-only attack such as the one used to defeat the simple substitution above; they had to resort to comparing known pairs of plaintext-ciphertext (called a "known plaintext attack") and even to baiting Enigma users into encrypting specific messages, and observing the result (a "chosen plaintext attack"). But this distinction was of little comfort to their enemies' defeated armies and sunk submarines.

From that high point in its history, frequency analysis just kind of faded away. Modern ciphers, inspired by the needs of the information age, were designed to operate with individual bits, not letters. More importantly, these ciphers were designed with a somber understanding of what later came to be known as Schneier's Law (https://www.schneier.com/blog/archives/2011/04/schneiers_law.html): anyone can create an encryption algorithm that they themselves can't break. It's not enough for the cipher machinery to *appear* complex — to prove its worth, it must resist a ruthless security review by many cryptanalysts doing their best to break the cipher.

Precomputation Attack



Consider the hypothetical city of Precom Heights, population 200,000. An apartment in Precom contains about $30,000 worth of valuables on average, and no more than $50,000. The security market in Precom is monopolized by ACME Industries, which produces the fabled Coyote Class(tm) door locks. According to expert analysis, the only thing that can break a Coyote Class lock is a "meeper" — a very complex hypothetical machine that would require an investment of about 5 years and $50,000 to construct. Is the city secure?

Probably not. Eventually an ambitious enough criminal will come along. They'll reason thus: "Yes, I'm eating a large up-front cost. Five years of waiting patiently, and $50,000 spent on top of that. But when the work is done, I'll have access to the *entire fortune of this city*. If I play my cards right, this investment will pay for itself many times over."

A similar dynamic applies in cryptography. Attacks against a specific cipher are subject to a ruthless cost-benefit analysis; if the analysis is not favorable, the attack won't happen. But attacks that apply to many potential victims almost always pay off, and if they do, the best design practice is to assume them to be going on from day one. We basically have a Murphy's Law of Cryptography: "Anything that feasibly *could* break the system, *will* break the system."

The simplest example of a cryptosystem vulnerable to a precomputation attack is one where the encryption algorithm is constant, and no key is used. This was the case with the Caesar Cipher (https://en.wikipedia.org/wiki/Caesar_cipher), which simply shifted each letter of the alphabet 3 letters ahead (looping around, so the last letter in the alphabet was encrypted as the third). Kerckhoffs' principle rears its head again; once the system is broken, it is broken forever.

Precomputation attacks are a simple concept. Even the most amateur cryptosystem designer is likely to see them coming, and prepare accordingly. As a result, if you look at the timeline of the evolution of cryptography, precomputation attacks were irrelevant through most of it — starting with the first improvements on the Caesar cipher, all the way up to the decline of polyalphabetic ciphers. These attacks only saw a comeback after the rise of the modern era of cryptography.

This comeback was fueled by two factors. First, cryptosystems finally emerged that were complex enough to contain "break once, use later" opportunities that weren't obvious. Second, cryptography reached such wide use that millions of laymen were making decisions every day about what pieces of cryptography to reuse, and where. It was a while until experts realized the resulting risks, and raised the alarm.

Keep precomputation attacks in mind; by the end of this article, we'll see two separate real-life cryptographic breaches where this type of attack played an important part.

Interpolation Attack

Here is the famous detective, Sherlock Holmes, performing an interpolation attack on the hapless Dr. Watson:

> I knew you came from Afghanistan. [..] The train of reasoning ran, 'Here is a gentleman of the medical type, but with the air of a military man. Clearly an army doctor, then. He has just come from the tropics, for his face is dark, and that is not the natural tint of his skin, for his wrists are fair. He has undergone hardship and sickness, as his haggard face says clearly. His left arm has been injured: He holds it in a stiff and unnatural manner. Where in the tropics could an English army doctor have seen much hardship and got his arm wounded? Clearly in Afghanistan.' The whole train of thought did not occupy a second. I then remarked that you came from Afghanistan, and you were astonished.

Holmes could extract very little information from any of the clues individually; he was only able to come to his conclusion by considering them all together. Similarly, an interpolation attack works by examining known pairs of plaintext and ciphertext, all derived from the same key; and from each pair, making a broad deduction about the key. The deductions are all vague and apparently useless, until suddenly they reach a critical mass and lead to a single conclusion that, however improbable, must be the truth. The key is revealed, or else the process of decryption is understood so thoroughly that it can be replicated.

We'll illustrate the way the attack works with a simple example. Suppose that we are attempting to read the private journal of our frenemy, Bob. Bob encrypts every number in his journal with a simple cryptosystem he's read about in a blurb in *Mock Crypto Magazine*. The system works as follows: Bob picks two numbers close to his heart M and N. From then on, to encrypt any number x, he computes $Mx + N$. For example, if Bob picked $M = 3$ and $N = 4$, then under encryption, 2 would become $3 * 2 + 4 = 10$.

Suppose on December 27, we witness Bob writing in his journal. When Bob is done, we discreetly pick up the journal and examine the latest entry:

> Date: 235/520
> Dear Diary,
> Today was a good day. In 64 days I have a date with Alice, who lives down at number 843. I really think she could be the 26!

Since we are really anxious to stalk Bob during his date (in this scenario we are 15 years old), we are interested in finding out the day of Bob's date, as well as Alice's address. Happily, we notice that the cryptosystem Bob is using is vulnerable to an interpolation attack. We may not know M and N, but we do know the date today, and therefore we have two plaintext-ciphertext pairs. To wit, we know that 12 encrypted is 235, and furthermore, that 27 encrypted is 520. We can therefore write:

$$M * 12 + N = 235$$

$$M * 27 + N = 520$$

Now, since we are 15 years old we also know that this is what's called "2 equations with 2 unknowns", and that in this situation, it is possible to solve for M and N without too much trouble. Each plaintext-ciphertext pair created a constraint on Bob's key, and the combined 2 constraints were enough to recover the key completely. In the example above, the solution is $M = 19$ and $N = 7$.

Interpolation attacks are, of course, not limited to such simple examples. Every cryptosystem that boils down to a well-understood mathematical object and a list of parameters is at risk of an interpolation attack — the better understood the object, the higher the risk.

People studying cryptography have been known to complain about it being "the art of designing things to be as ugly as possible", and interpolation attacks probably carry much of the blame for this. Bob can either have a cryptosystem with mathematically elegant design, or he can have privacy on his date with Alice — but alas, he typically cannot have both. This will become startlingly clear when we eventually get to the subject of public-key cryptography.

Cross Protocol / Downgrade Attack

In the 2013 film *Now You See Me*, an entourage of stage magicians called the "Horsemen" endeavor to swindle corrupt insurance mogul Arthur Tressler out of his entire fortune. To access Arthur's bank account, the Horsemen have to either present his username and password, or have him show up at the bank in person and cooperate with their scheme.

Both of these are very difficult feats; the Horsemen are stage magicians, not the Mossad. So, instead, they target a third possible protocol — they have an accomplice call the bank and pretend to be Arthur. The bank asks for several personal details to verify Arthur's identity, such as his Uncle's name and his first pet's name; the Horsemen extract this information from Arthur in advance easily, via deft social engineering (https://www.youtube.com/watch?v=95jHwnAhHgU). At that point, the excellent security of the password does not matter any more.

(According to an urban legend that we have verified independently, cryptographer Eli Biham was once confronted by a bank teller who insisted on installing password recovery questions of this type. When the teller asked Biham for the latter's maternal grandmother's name, Biham started reading out: "Capital X, small y, three, ...")

Similarly to the above, it sometimes happens that two cryptographic protocols are employed side-by-side to secure the same asset, while one protocol is much weaker than the other. The resulting setup is then vulnerable to a cross-protocol attack, where features in the weaker protocol are abused in order to compromise the stronger protocol.

In some more complicated cases, the attack can't succeed just by contacting a server with the weaker protocol, and requires the unwitting participation of a legitimate client. This can still be arranged using something called a downgrade attack. To understand how such an attack works, suppose the Horsemen were dealing with a more difficult challenge than the one in the movie; specifically, suppose the Bank teller and Arthur had some contingencies in place, resulting in this dialogue:

> ATTACKER: Hello? This is Arthur Tressler. I would like to recover my password.
>
> TELLER: Excellent. Please look at your personally issued secret code book, page 28, word 3. All the following communication will be encrypted with this specific word as the key. `PQJGH. LOTJNAM PGGY MXVRL ZZLQ SRIU HHNMLPPPV`...

/

ATTACKER: Wait, wait, wait. Is this really necessary? Can't we just speak to each other like normal human beings?

TELLER: I advise against it.

ATTACKER: I'm just — listen, I've had a lousy day, okay? I'm a paying customer, and I am not in the mood for fancy complicated code books.

TELLER: Fine. If you insist, Mr. Tressler. What is your request?

ATTACKER: I would like to please transfer all my money to the Victims of Arthur Tressler National Fund.

(There is a pause.)

TELLER: I see. Please provide your large transaction PIN code.

ATTACKER: My what now?

TELLER: Per your personal request, transactions of this magnitude require that you provide your large transaction PIN code. This code was issued to you when you first opened your account.

ATTACKER: ...I've lost the code. Is this really necessary? Can't you just approve the transaction?

TELLER: No. Apologies, Mr. Tressler. Again, this is a security measure you requested. We can issue you a new PIN, sent to your PO box, if you'd like.

The Horsemen ponder the challenge for a long while. They listen in on several of Tressler's big transactions, hoping to hear the PIN; but every time, the conversation turns into encrypted gibberish before they can hear anything interesting. Finally, one day, they put a plan in motion. They patiently wait until Tressler has to make a large transaction by phone, tap into the line, and then...

TRESSLER: Hello. I would like to issue a remote transaction, please.

TELLER: Excellent. Please look at your personally issued secret code book, page –

(The ATTACKER presses a button; the TELLER's voice turns into indecipherable noise.)

TELLER: — #@$#@$#&#*@$$@#* will be encrypted with this word as the key. `AAAYRR PLRQRZ MMNJK LOJBAN` –

TRESSLER: Sorry, I didn't quite catch that. Come again? What page? What word?

TELLER: It's page @#$@#*$)#*#@()#@$(#@*$(#@*.

TRESSLER: What?

TELLER: Word number twenty @$#@$#%#$.

TRESSLER: Seriously! Come off it! You and your security protocol are a blight. I KNOW you can just normally talk to me.

TELLER: I advise against —

TRESSLER: I advise you stop wasting my time. I don't want to hear any more of it until you fix your phone line issues. Can we do this transaction or not?

TELLER: ...yes. Fine. What is your request?

TRESSLER: I would like to transfer $20,000 to Lord Business Investments, account number –

TELLER: A moment please. That is a large transaction. Please provide your large transaction PIN.

TRESSLER: What? Oh, right. It is 1234.

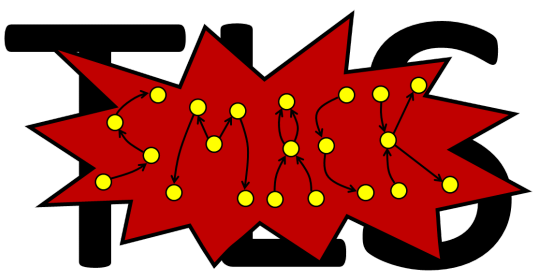
And that's a downgrade attack. The weaker protocol, "just speak plainly", was supposed to be an *optional* addition to be used as a last resort. And yet, here we are.

You might wonder who in their right mind would design a real-world system analogous to a "secure, unless you come in sideways" system, or a "secure, unless you insist otherwise" system, as described above. But much like the fictional bank would rather take the risk and retain its crypto-averse customers, systems in general often bow to requirements that are indifferent, or even overtly hostile, to security needs.

Exactly such a story surrounded the release of SSL protocol version 2 in the year 1995. The United States government had long since come to view cryptography as a weapon, best left out of the hands of geopolitical enemies and domestic threats. Pieces of code were approved on a case-by-case basis for leaving the US, often conditional on the algorithm being weakened deliberately. Netscape, then the main vendor of web browsers, was able to obtain a permit for SSLv2 (and by extension, Netscape Navigator) to support a vulnerable-by-design RSA with a key length of 512 bits (and similarly 40 bits for RC4).

By the turn of the millennium, regulations had been relaxed and access to state-of-the-art encryption became widely available. Still, clients and servers supported export-grade crypto for years, due to the same inertia that preserves support for any legacy system. Clients figured they might encounter a server which doesn't support anything else, so they hung on to optional support for it, as a last resort. Servers did the same. Of course, SSL protocol dictates that clients and servers should never use a weak protocol when a better one is available — but then again, neither should Tressler and his bank.

The theory we have discussed so far came to a head in two consecutive high-profile attacks that rattled the security of SSL protocol in 2015, each discovered by researchers at Microsoft and INRIA (https://www.inria.fr/en/). First was the FREAK attack, announced in February of that year; three months later it was followed by another similar attack called Logjam, which we'll discuss in more detail when we get to attacks on public-key cryptography.



The FREAK (https://mitls.org/pages/attacks/SMACK) attack (also "Smack TLS") resulted when the research team analyzed TLS client/server implementations and found a curious bug. In these implementations, if the client never asks to use export-grade weak cryptography, but the server responds with such keys anyway — the client says "oh well" and complies, carrying out the entire conversation using the weak cipher suite.

At the time, public perception on export-grade cryptography held that deprecated equals irrelevant; the attack came as quite a shock, and affected many high-profile domains — including some belonging to the White House, the IRS and the NSA. Worse, it turned out that many vulnerable servers had a performance optimization where the same keys were used over and over, instead of new keys being generated for each session. This allowed a precomputation attack on top of the downgrade attack: breaking a single key was still somewhat costly ($100 and 12 hours at the time of the publication), but the practical cost of the attack per connection was drastically lower. The key could be broken once, and then the break used for every connection made by the server from that point on.

And one advanced attack we need to know before we go any further...

Oracle Attack

Moxie Marlinspike (https://moxie.org/) may be best known as the father of the cross-platform encrypted messaging service, Signal (https://signal.org/); but personally, we are fond of one of his lesser-known innovations — the cryptographic doom principle (https://moxie.org/blog/the-cryptographic-doom-principle/). Slightly paraphrased, it states: "If a protocol performs *any* cryptographic operation on a message with a possibly malicious origin, and behaves differently based on the result, this will inevitably lead to doom." Or, put more abruptly — "Don't chew on enemy input, and if you must, at least don't spit any of it back out."

Never mind the buffer overflows, command injections and the like; they're beyond the scope of this discussion. Violating the doom principle leads to fair and square cryptographic breaks, which result from the protocol behaving exactly like it was supposed to.

To demonstrate how, we'll present a toy setup — based on a simple substitution cipher — which violates the doom principle; and then demonstrate an attack made possible by the violation. While we've already seen an attack on the simple substitution cipher based on frequency analysis, this isn't merely "another way to break the same cipher." To the contrary: oracle attacks are a much more modern invention, applicable to plenty of situations where frequency analysis will fail, and we'll see a demonstration of this in the next section. This simpler cipher was picked just to make the exposition smoother.

On with the example. Alice and Bob communicate using a simple substitution cipher, using a key known only to them. They are very strict with message lengths, and only willing to deal with messages that are exactly 20 characters long. Therefore, they've agreed that if someone wants to send a shorter message, they have to append some dummy text to the end of the message to get it to be exactly 20 characters. After some discussion, they've decided they will only accept the following dummy texts: `a`, `bb`, `ccc`, `dddd`, and so forth and so on. That way, there is an available dummy text of every possible required length.

When Alice or Bob receive a message, after decrypting it, they first check that the plaintext is the proper length (20 characters), and the suffix is a proper dummy text. If it isn't, they reply with an appropriate error message. If the text length and dummy text are both OK, the recipient reads the message itself and sends an encrypted reply.

The attack proceeds by impersonating Bob, and sending forged messages to Alice. The messages are complete nonsense — the attacker does not have the key, and so cannot forge a meaningful message. But since the protocol violates the doom principle, the attacker can still bait Alice into disclosing information about the key, as follows.

> ATTACKER: `PREWF ZHJKL MMMN. LA`
>
> ALICE: Incorrect dummy text.
>
> ATTACKER: `PREWF ZHJKL MMMN. LB`
>
> ALICE: Incorrect dummy text.
>
> ATTACKER: `PREWF ZHJKL MMMN. LC`
>
> ALICE: `ILCT? TLCT RUWO PUT KCAW CPS OWPOW!`

(The attacker has no idea what Alice just said, but notes that `C` must map to `a`, since Alice accepted the dummy text.)

> ATTACKER: `REWF ZHJKL MMMN. LAA`
>
> ALICE: Incorrect dummy text.
>
> ATTACKER: `REWF ZHJKL MMMN. LBB`
>
> ALICE: Incorrect dummy text.
>
> (Some trials later...)
>
> ATTACKER: `REWF ZHJKL MMMN. LGG`
>
> ALICE: Incorrect dummy text.
>
> ATTACKER: `REWF ZHJKL MMMN. LHH`
>
> ALICE: `TLQO JWCRO FQAW SUY LCR C OWQXYJW. IW PWWR TU TCFA CHUYT TLQO JWFCTQUPOLQZ.`

(The attacker, again, has no idea what Alice just said, but notes that `H` must map to `b`, since Alice accepted the dummy text.)

And so on, until the attacker knows the plaintext counterpart of every letter.

This may appear superficially similar to a chosen ciphertext attack. After all, the attacker gets to choose ciphertexts and the server dutifully processes them. The major difference, which makes attacks like these viable in the real world, is that the attacker does not require access to the actual decryption — the server's response is enough, even something as innocuous as "incorrect dummy text."

While it's instructive to understand how this specific attack took place, one shouldn't get too hung up on the specifics of the "dummy text" scheme, the specific cryptosystem used, or the exact sequence of messages sent by the attacker. The main idea here is how Alice reacts differently based on properties of the plaintext, and does so without verifying that the corresponding ciphertext really originated with a trusted party. By doing so, Alice makes it possible for an attacker to squeeze secret information out of her responses.

We could change many things about the scenario, such as the plaintext property that triggers the difference in Alice's behavior, or the difference in behavior itself, or even the cryptosystem used — but the principle would remain the same, and the attack would generally remain viable, in one form or another. This dawning realization was responsible for the discovery of several security bugs, which we'll delve into in a moment; but before that could happen, some theoretical seeds had to be planted. How do we take this toy "Alice Scenario" and mold it into an attack that can work on an actual modern cipher? Is that possible at all, even in theory?

In 1998, Swiss cryptographer Daniel Bleichenbacher answered that question in the positive. He demonstrated an oracle attack against the widely-used public-key cryptosystem, RSA, when used with a certain message scheme. In some RSA implementations, the server replied with a different error message, depending on whether the plaintext matched the scheme or not; this was enough to enable the attack.

Four years later, in 2002, French cryptographer Serge Vaudenay demonstrated an oracle attack almost identical to the one in the Alice scenario above — except instead of a toy cipher, he broke a whole respectable class of modern ciphers that people actually use. Specifically, Vaudenay's attack targeted ciphers with a fixed input size ("block ciphers") when used in a specific way called the "CBC mode of operation" and with a certain popular padding scheme basically equivalent to the one in the Alice scenario.

Also in 2002, American cryptographer John Kelsey — co-author of Twofish (https://en.wikipedia.org/wiki/Twofish) — proposed a variety of oracle attacks on systems that compress messages and then encrypt them. The most notable among those was an attack that took advantage of the fact that it is often possible to tell the original plaintext length from the ciphertext length. This, in theory, enabled an oracle attack that recovers portions of the original plaintext.

We follow with a more detailed exposition of Vaudenay's and Kelsey's attacks (we'll give a more detailed exposition of Bleichenbacher's attack when we get to attacks on public-key cryptography). The text gets somewhat technical, despite the best of our efforts; so if the above is enough detail for you, skip down past the following two sections.

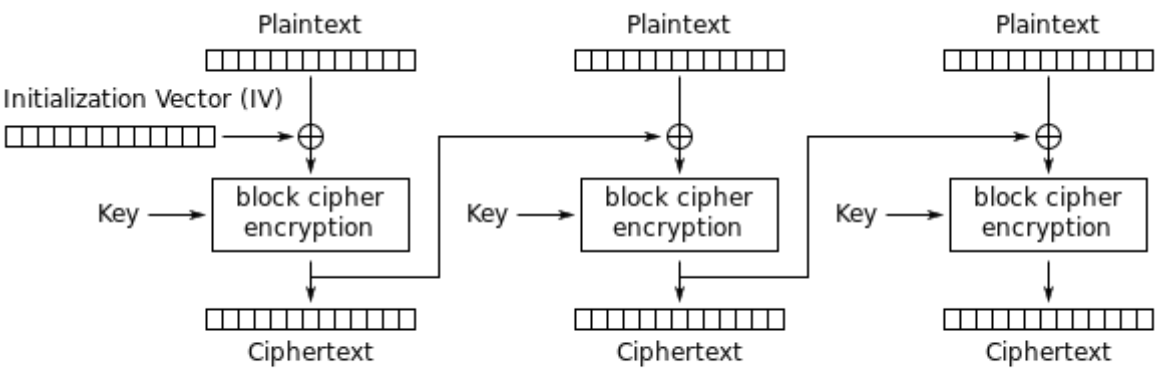
Vaudenay's Attack

To understand Vaudenay's attack, we first need to talk about block ciphers and modes of operation in a little bit more detail. A "block cipher" is, as mentioned, a cipher which takes a key and input of a certain fixed length (the "block length"), and outputs an encrypted block of the same length. Block ciphers are widely used and are considered relatively secure. The now-retired DES, widely considered the first modern cipher, was a block cipher; as mentioned above, the same is true for AES, which is in extensive use today.

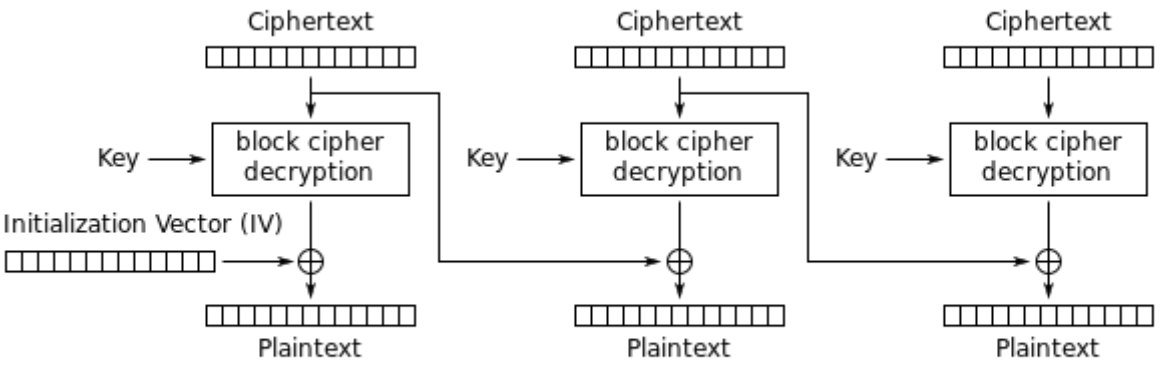
Unfortunately, block ciphers have one glaring weakness. The typical block size is 128 bits, or 16 characters. Obviously modern uses of cryptography require that we work with inputs longer than that, and this is where modes of operation come in. A mode of operation is basically a hack — an algorithm for taking a block cipher, which can only take a fixed amount of input, and somehow applying it to inputs of arbitrary lengths.

Vaudenay's attack targets a popular mode of operation, called CBC (Cipher Block Chaining). The attack treats the underlying block cipher as a magical unassailable black box, and bypasses its security entirely.

Here is a diagram that illustrates how CBC mode operates:



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

The circled plus signs stand for XOR operations. So, for example, the second ciphertext block is obtained by:

1. XORing the second plaintext block with the first ciphertext block.
2. Encrypting the resulting block with the block cipher, using the key.

As CBC makes such heavy use of the XOR operation, let us take a moment to recall its following useful properties.

- Identity: $A \oplus 0 = A$
- Commutativity: $A \oplus B = B \oplus A$
- Associativity: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$

- Involution: $A \oplus A = 0$
- Bytewise: Byte n of $(A \oplus B)$ = (Byte n of A) $\oplus$ (Byte n of B)

These properties imply, as a rule of thumb, that if we have an equation involving XORs and one unknown, it's possible to solve for the unknown. For example, if we know that $A \oplus X = B$ with X the unknown and A, B known, we can rely on the properties above to solve for X. XORing both sides of the equation with A, we obtain $X = A \oplus B$. This will all become very relevant in a moment.

There are two minor differences, and one major difference, between the Alice scenario we saw in the last section and Vaudenay's attack. The two minor differences are:

- In the Alice scenario, Alice expected plaintexts to end with `a`, `bb`, `ccc` and so on. In Vaudenay's attack, the victim instead expects plaintexts to end with N times the byte N (that is, hexadecimal `01`, or `02 02`, or `03 03 03` and so on). This difference is purely cosmetic, and has little practical effect.
- In the Alice scenario, it was easy to tell whether Alice accepted the message, based on the "incorrect dummy text" response. In Vaudenay's attack, the analysis is more involved, and depends on the exact implementation attacked; but for the sake of brevity, take it as a given that this analysis is still possible.

The one major difference is:

- As we're not using the same cryptosystem, the relationship between the attacker-controlled ciphertext bytes and the unknowns (key and plaintext) is obviously different. The attacker therefore has to use another strategy when crafting ciphertexts and interpreting server responses.

This last difference is the final piece missing to understand Vaudenay's attack, so let's take a moment to think about why and how it should be possible to mount an oracle attack on CBC at all.

Suppose we are given a CBC ciphertext composed of (let's say) 247 blocks, and we want to decrypt it. We can send forged messages to the server, just as earlier we were able to send forged messages to Alice. The server will decrypt messages for us, but will not provide us with the decryption — instead, again like with Alice, the server will tell us whether the resulting plaintext has valid padding or not.

Consider that in the Alice scenario, we had the following relationship:

$$\mathrm{SIMPLE\_SUBSTITUTION}(\mathrm{ciphertext}, \mathrm{key}) = \mathrm{plaintext}$$

Let's call this the "Alice equation." We controlled the ciphertext; the server (Alice) leaked vague information about the resulting plaintext; and this allowed us to deduce information about the remaining term — the key. By analogy, it stands to reason that if we can find a similar relationship for the CBC scenario, we might be able to extract some secret information there, too.

Happily, a relationship does exist for us to exploit. Consider the output of the final invocation of the "block cipher decryption" box, and denote that output W. Also denote plaintext blocks $P_1, P_2, \ldots$, and ciphertext blocks $C_1, C_2, \ldots$. Take a look again at the CBC diagram, and note that we have:

$$C_{246} \oplus W = P_{247}$$

Let's call this the "CBC equation."

In the Alice scenario, by controlling the ciphertext and watching Alice leak information about the corresponding plaintext, we were able to mount an attack that recovered the third term in the equation — the key. In the CBC scenario, we also control the ciphertext and observe information leaks regarding the corresponding plaintext. If the analogy carries, we should be able to gain information about W.

Suppose we do recover W; what then? Well, we can then immediately deduce the entire last block of plaintext (P_{247}) simply by plugging in C_{246} (which we have) and W (which we would also have) into the CBC equation.

So, we have an optimistic intuition about a general outline for an attack, and it's time to work out the details. We turn our attention to the exact manner in which the server leaks information about the plaintext. In the Alice scenario, the leak resulted from Alice responding with a proper message if and only if $\mathrm{SIMPLE\_SUBSTITUTION}(\mathrm{ciphertext}, \mathrm{key})$ ended in the string `a` (or `bb`, et cetera, but the chances of randomly triggering these conditions were very small). Similarly with CBC, the server accepts the padding if and only if $C_{246} \oplus W$ ends in hexadecimal `01`. So let's try the same trick — sending forged ciphertexts, with our own forged values of C_{246}, until the server accepts the padding.

When the server does accept the padding for one of our forged messages, this implies that:

$$C_{246} \oplus W = \text{something ending in hex 01}$$

We now use the bytewise property of XOR:

$$(\text{final byte of } C_{246}) \oplus (\text{final byte of } W) = \text{hex 01}$$

We know both the first and third term, and we have already seen that this allows us to recover the remaining term – the final byte of W:

$$(\text{Final byte of } W) = (\text{Final byte of } C_{246}) \oplus (\text{hex 01})$$

This also gives us the final byte of the final plaintext block via the CBC equation and the bytewise property.

We might call the attack off now and be content that we've succeeded in doing something that should be impossible. But actually, we can do much better: we can recover the entire plaintext. This does require a certain trick that did not appear in the original Alice scenario and is not a necessary feature of oracle attacks — but the method is worthwhile to understand all the same.

To see how this larger feat might be accomplished, first note that by deducing the correct value of the last byte of W, we have also gained a new ability. From now on, when we forge ciphertexts, we can control the last byte of the corresponding plaintext. This is, again, due to the CBC equation and the bytewise property:

$$(\text{final byte of } C_{246}) \oplus (\text{final byte of } W) = \text{Final byte of } P_{247}$$

Since we now know the second term, we can use our control of the first term to control the third. We just compute:

$$(\text{Final byte of forged } C_{246}) = (\text{Desired final byte of } P_{247}) \oplus (\text{Final byte of } W)$$

Earlier we couldn't do this, because we didn't yet have the final byte of W.

How does this help us? Suppose we now rig all our future ciphertexts so that in the corresponding plaintexts, the final byte is `02`. The server will now only accept the padding if the plaintext ends with `02 02`. As we fixed the last byte, this will only happen if the second-to-last plaintext byte is also `02`. We keep sending forged ciphertext blocks, varying the second-to-last byte, until the server accepts the padding for one of them. At that point we have:

$$(\text{Second-to-final byte of forged } C_{246}) \oplus (\text{Second-to-final byte of } W) = \text{hex 02}$$

And we recover the second-to-final byte of W in exactly the same way that we recovered the final byte earlier. From there, we continue in the same fashion. We fix the last two plaintext bytes to `03 03` and repeat this same attack for the third-to-last byte, and so on, eventually recovering W in its entirety.

What about the rest of the plaintext? Well, note that the value W that we have recovered is actually $\text{BLOCK\_DECRYPT}(\text{key}, C_{247})$. We could have put any other block there instead of C_{247}, and the attack would have still been successful. Effectively, we can get the server to BLOCK_DECRYPT anything for us. At that point, it's game over – we can decrypt any ciphertext we want (take another look at the CBC decryption diagram to become convinced of this; and note that the IV is public).

This particular technique of bootstrapping to a block-decryption oracle is worth paying attention to, as it plays a crucial role in an attack we'll come across later.

Kelsey's Attack

Kelsey, a man after our own hearts, outlined the principles behind many different possible attacks, rather than the finer details of one specific attack on one specific cipher. His 2002 paper (https://www.iacr.org/cryptodb/archive/2002/FSE/3091/3091.pdf) is a study of possible attacks on encrypted compressed data. You'd think that to mount an attack, you'd need more to go on than "the data was compressed and then encrypted", but apparently that's enough.

This surprising result is due to two principles at work. First, there tends to be a strong correlation between plaintext length and ciphertext length; for many ciphers, these two are exactly equal. Second, when compression is performed, there tends to also be a strong correlation between the compressed length and the degree to which the original text was "noisy" and non-repetitive (the technical term is "high-entropy").

To see this principle in action, consider the following two plaintexts:

Plaintext 1: `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`

Plaintext 2: `ATVXCAGTRSVPTVVULSJQHGEYCMQPCRQBGCYIXCFJGJ`

Suppose both these plaintexts are compressed, then encrypted. You are given the two resulting ciphertexts, possibly out of order, and must guess which ciphertext corresponds to which plaintext:

Ciphertext A: `PVOVEYBPJDPVANEAWVGCIUWAABCIYIKOOURMYDTA`

Ciphertext B: `DWKJZXYU`

The answer is clear. Among the plaintexts, only plaintext 1 could have been compressed to the meager length of ciphertext B. We figured this out without knowing anything about the compression algorithm, the cipher key or even the cipher itself; compared to the hierarchy of possible cryptographic attacks, that's kind of insane.

Kelsey goes on to point out that, under certain unusual circumstances, this principle could also be used to launch oracle attacks. To be specific, he outlines how an attacker can recover a secret plaintext if they can get the server to compress-then-encrypt data of the form (plaintext followed by X), as long as the attacker controls X, and can somehow observe the length of the encrypted result.

Again, as in other oracle attacks, we have a relationship:

$$\text{Encrypt}(\text{Compress}(\text{Plaintext followed by} X)) = \text{Ciphertext}$$

Again, we control one term (X), receive a small information leak about another term (the ciphertext), and seek to recover the remaining term (the plaintext). Despite the valid analogy, you'll note that this is an unusual setup, compared to the other oracle attacks we've seen.

To illustrate how such an attack might work, we use a toy compression scheme that we just made up: TOYZIP. TOYZIP looks for strings of text that already appear earlier in the text, and replaces them with 3 "placeholder" bytes that indicate where to find the earlier instance of the string, and how long it is. So, for example, the string `helloworldhello` might be compressed to `helloworld[00][00][05]`, which has a length of 13 bytes compared to the original's 15.

Suppose an attacker is trying to recover a plaintext of the form `password=...`, where the password itself is unknown. In line with Kelsey's attack model, the attacker can ask the server to compress-then-encrypt messages of the form (plaintext followed by X), where X is any text of the attacker's choice. When the server is done, it reports the length of the result. The attack proceeds as follows:

Attacker: Please compress & encrypt the plaintext with no additions.

Server: Result has length 14.

Attacker: Please compress & encrypt the plaintext, followed by `password=a`.

Server: Result has length 18.

(Attacker notes: [original 14] + [3 bytes that replaced `password=`] + a)

Attacker: Please compress & encrypt the plaintext followed by `password=b`.

Server: Result has length 18.

Attacker: Please compress & encrypt the plaintext followed by `password=c`.

Server: Result has length 17.

(Attacker notes: [original 14] + [3 bytes that replaced `password=c`]. This implies that the original plaintext contains the string `password=c`. Meaning, the password starts with the letter c.)

Attacker: Please compress & encrypt the plaintext followed by `password=ca`.

Server: Result has length 18.

(Attacker notes: [original 14] + [3 bytes that replaced password=c] + a)

Attacker: Please compress & encrypt the plaintext followed by `password=cb`.

/

> **Server:** Result has length 18.
>
> (...some time later...)
>
> **Attacker:** Please compress & encrypt the plaintext followed by `password=co`.
>
> **Server:** Result has length 17.
>
> (Attacker notes: [original 14] + [3 bytes that replaced `password=co`]. By the same logic that gave us the first letter, the password must start with the letters `co`.)

And so forth and so on, until the whole password is recovered.

The reader would be forgiven for thinking that this is a purely academic exercise, and that such an attack scenario would never arise in the real world. Alas, as we'll see in a short moment, it's best to never say "never" in cryptography.

Brand-Name Vulnerabilities: CRIME, POODLE, DROWN

Finally, after persevering through all the above theory, we can now see how these principles of offense played out in real-world cryptographic vulnerabilities.

CRIME

When you are an attacker preying on a victim's browser and network, some things are supposed to be easy, and others difficult. For instance, seeing the victim's web traffic is easy; it's enough for the two of you to be seated at the same Starbucks. For this reason, it's usually recommended for potential victims (i.e. everyone) to use an encrypted connection. Less easy, but still possible, is making HTTP requests on the victim's behalf to some third-party site (e.g. Google). The attacker has to entice the victim into visiting a malicious webpage, which contains a script that will make the request. The web browser will automagically endow the request with the appropriate session cookie.

This might seem surprising; it apparently implies that if Bob visits `evil.com`, a script on that website can just ask Google to email Bob's password to `attacker@evil.com`. Can that really happen? Well, yes in theory, but actually no in practice. That scenario is called a Cross-Site Request Forgery (https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) attack (CSRF), and it was more relevant around the mid-nineties. Today, if `evil.com` tries that trick, Google (or any self-respecting website) will typically respond: "excellent, but before we proceed, your CSRF token for this transaction is... mmm... `three trillion and seven`. Please repeat that back to me." Modern browsers enforce something called the "same-origin policy", according to which scripts running on website A do not get to access information sent by website B. The `evil.com` script can therefore send requests to `google.com`, but not read any responses, or actually complete a transaction.

We should stress that if Bob is not using an encrypted connection, all of these defenses are meaningless. The attacker can simply read Bob's traffic and recover the Google session cookie. Armed with the cookie, the attacker can just start a new Google tab from the comfort of their own browser, and impersonate Bob without having to deal with pesky same-origin policies. But, unfortunately for the attacker, that's a pretty big if. The internet at large has long since declared war on plain-text connections, and Bob's outgoing traffic is probably encrypted, whether he likes it or not. In fact, back in the day, the traffic would also first be *compressed* before being encrypted; this was routine practice by web clients to improve latency.

Enter CRIME (https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222). The initials stand for Compression Ratio Infoleak Made Easy, and the attack was unveiled in September of 2012 by security researchers Juliano Rizzo and Thai Duong. We already have all the pieces in place to understand what they were able to pull off, and how. An attacker can make Bob's browser send requests to Google, and then eavesdrop on the LAN to recover the resulting requests in their compressed, encrypted form. We therefore have:

$$\text{Web traffic} = \text{Encrypt}(\text{Compress}(\text{request followed by cookie}))$$

Where the attacker controls the request, and has access to the sniffed web traffic in its entirety, including its length. Kelsey's pipe-dream scenario had come to life.

Based on this insight, the authors of CRIME crafted an exploit that could steal session cookies associated with a wide variety of sites, including Gmail, Twitter, Dropbox and Github. CRIME affected most modern web browsers, and in response to this attack, patches had to be issued that silently buried the feature of SSL compression, never again to see the light of day. The one notable exception was the venerable Internet Explorer, which had never implemented the feature in the first place.

POODLE

In October of 2014, a Google security team caused great alarm by exploiting a vulnerability in the SSL protocol that had already been patched for over a decade.

It turned out that while servers were running the shiny and updated TLSv1.2, many of them included support for the antiquated SSLv3, put there for the sake of backwards compatibility with Internet Explorer 6. We've already spoken about downgrade attacks, so you can already see where this is going. A bit of well-placed sabotage of the protocol handshake, and these servers were eager to fall back on the old SSLv3 protocol, effectively setting back the security clock by 15 years.

To put this in its proper historical context, here's Matthew Green with a short summary of the history of SSL up until version 2 (https://blog.cryptographyengineering.com/2016/03/01/attack-of-week-drown/):

> Transport Layer Security (TLS) is the most important security protocol on the Internet. [..] nearly every transaction you conduct on the internet relies on TLS. [..] But TLS wasn't always TLS. The protocol began its life at Netscape Communications (https://en.wikipedia.org/wiki/Netscape) under the name "Secure Sockets Layer", or SSL. Rumor has it that the first version of SSL was so awful that the protocol designers collected every printed copy and buried them in a secret New Mexico landfill site. As a consequence, the first public version of SSL is actually SSL version 2 (http://www-archive.mozilla.org/projects/security/pki/nss/ssl/draft02.html). It's pretty terrible as well [..] it was a product of the mid-1990s, which modern cryptographers view as the "dark ages of cryptography (https://blog.cryptographyengineering.com/2012/09/on-provable-security-of-tls-part-1.html)". Many of the nastier cryptographic attacks we know about today had not yet been discovered. As a result, the SSLv2 protocol designers were forced to essentially grope their way in the dark, and so were frequently devoured by grues (https://www.schneier.com/cryptography/paperfiles/paper-ssl.pdf) – to their chagrin and our benefit, since the attacks on SSLv2 offered priceless lessons for the next generation of protocols.

Following these events, in 1996, a disillusioned Netscape redesigned the SSL protocol from the ground up. The result was SSL version 3, which fixed several of its predecessor's known security issues (https://stason.org/TULARC/security/ssl-talk/4-11-What-is-the-difference-between-SSL-2-0-and-3-0.html).

Happily for attackers, "several" does not mean "all." Broadly speaking, SSLv3 included all the necessary building blocks to launch Vaudenay's attack. The protocol performed encryption using a block cipher in CBC mode, and used a padding scheme that was not designed with security in mind (this was fixed when SSL became TLS; hence the need for the downgrade attack). If you'll recall the padding scheme we discussed in our original description of Vaudenay's attack, the scheme used by SSLv3 was pretty similar.

But, unhappily for attackers, "similar" does not mean "identical." SSLv3's padding scheme is of the form (N arbitrary bytes followed by the number N). Try to pick an imaginary ciphertext block and work through the stages of Vaudenay's original method under these conditions; you'll find that the attack does successfully extract the rightmost byte out of the corresponding plaintext block, but cannot proceed past that initial win. Decrypting every 16th byte of a ciphertext is a nifty parlor trick, but it's not capital-V Victory.

Faced with this setback, the Google team opted for a solution of last resort: they switched to a more powerful threat model — the one used in the CRIME attack. If we assume the attacker is a script running inside the victim's browser tab, and prove it can extract the victim's session cookie, that's still a respectable feat. While it's true that a more powerful threat model is a less feasible one, we've already seen in the previous section that this specific model is feasible enough.

Given this more capable adversary, the attack can now finally proceed. Consider that the attacker knows where the encrypted session cookie appears in the header, and controls the length of the HTTP request that precedes it. They can, therefore, manipulate the HTTP request so that the final byte of the session cookie is aligned with the end of a block. That byte is now ripe for decryption. When that's done, the attacker can simply add a single character to the request; now the second-to-final byte of the session cookie will sit in the same spot, and be ripe for picking using the same method. The attack continues in this fashion until the cookie is recovered in full. That's POODLE (https://www.openssl.org/~bodo/ssl-poodle.pdf) – the Padding Oracle on Downgraded Legacy Encryption.

DROWN

As we've touched on previously, SSLv3 may have had its kinks, but it had nothing on its predecessor; SSLv2 protocol, a product of a different era. Attacks that were possible against it are now included in the security 101 syllabus. Victims would have their messages cut in mid-sentence, with `I'll agree to that over my dead body` turning into `I'll agree to that`; client and server would meet online, grow to trust each other, exchange secrets, and then find out that they were both catfished by some malicious agent who impersonated each one in front of the other. Then there was the issue of export-grade cryptography, which we covered earlier during the exposition of FREAK. It was cryptographic Sodom and Gomorrah.

In March of 2016, a team of researchers from diverse technical backgrounds came together to make a startling realization: for security purposes, SSLv2 was still not dead. Yes, attackers could no longer downgrade modern TLS sessions to SSLv2, as that hole had been patched in the aftermath of FREAK and POODLE, but they could still approach servers and initiate SSLv2 sessions of their own.

You might ask, what do we care if they do? They'll have a vulnerable session, but this shouldn't affect other sessions, or the security of the server — right? Well, yes and no. Yes — that's how it should be in theory. No — because obtaining valid SSL certificates is a bother and a financial burden, resulting in many servers using the same certificates, and by extension the same RSA keys, for both TLS and SSLv2 connections. To make matters worse, due to a bug, the "disable SSLv2" option did not actually work in OpenSSL, a popular SSL implementation.

This enabled a cross-protocol attack on TLS, called DROWN (https://drownattack.com/) (Decrypting RSA with Obsolete and Weakened eNcryption). Recall that this is not the same thing as a downgrade attack; the attacker does not need to act as a "man in the middle", and the client does not need to be manipulated into participating in an insecure session. The attackers, at their leisure, initiate an insecure SSLv2 session with the server, attack the weak protocol and recover the server's private RSA key. This key is also valid for TLS connections, and at that point, all the security of TLS won't save it from being compromised.

To seal the deal, attackers still needed a working attack against SSLv2 that allowed them to recover not only some specific communication, but the server's private RSA key. While this is a tall order, they could take their pick from any attack that was fully mitigated later than the release of SSLv2, and eventually found an attack to suit their needs: Bleichenbacher's attack, which we had mentioned in passing earlier, and of which we'll later see a full technical exposition. Both SSL and TLS contain counter-measures to obstruct Bleichenbacher's attack, but some incidental features of SSL, combined with the short keys used in export-grade cryptography, made a version of the attack possible (https://blog.cryptographyengineering.com/2016/03/01/attack-of-week-drown/).

At the time of its publication, DROWN affected the servers of about a quarter of the top million domains, and was possible to implement with modest resources, more in the ballpark of mischievous individuals than nation-states. Extracting a server's RSA key was possible using an investment of eight hours and $440, and SSLv2 went from "deprecated" to "radioactive."

Wait, what about Heartbleed?

That's not a cryptographic attack in the same sense of the other attacks we've seen here; it's a buffer overread (https://xkcd.com/1354).

Let's take a break

We started off by introducing some basic maneuvers: brute-force, interpolation, downgrade, cross-protocol and precomputation. This was followed by a single advanced technique, perhaps the most salient ingredient in modern cryptographic offense: the oracle attack. We spent quite a while with the oracle attack, understanding not only the principle behind it, but also the technical details behind two specific instances: Vaudenay's attack on the CBC mode of operation, and Kelsey's attack on compress-then-encrypt protocols.

During our survey of Downgrade and Precomputation, we gave a short exposition of the FREAK attack, which made use of both these principles, as targeted websites were reduced to using weak keys, and then on top of that, opted to use the same keys again and again. We saved for later the full exposition of the (very similar) Logjam attack, which targeted a public-key algorithm.

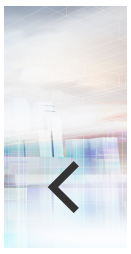
We then saw three more examples of cryptographic attack principles put into action. First, we took stock of CRIME and POODLE: two attacks which relied on an attacker's ability to inject plaintext side-by-side with the targeted plaintext, then view the server's response to the result, and *then* — using oracle attack methodology — pivot off this meager information to recover

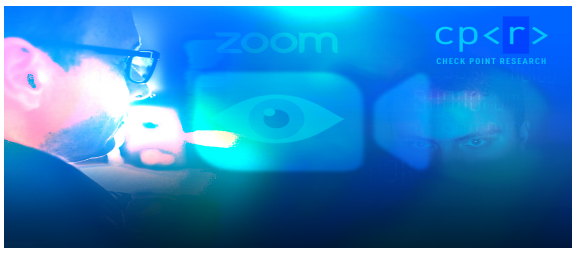
parts of the plaintext. CRIME went the route of Kelsey's attack on SSL compression, while POODLE instead used a variant of Vaudenay's attack on CBC to achieve the same effect.

We then turned out attention to DROWN — a cross-protocol attack which spoke to servers in obsolete SSLv2, then recovered their private encryption keys using Bleichenbacher's attack. For the time being, we skipped the technical details of that attack; like Logjam, it'll have to wait until we are more comfortable with public-key encryption and its attack landscape.

In the next blog post of this series, we'll talk about advanced attacks — such as meet-in-the-middle, differential cryptanalysis, and the birthday attack. We'll take a short foray into the land of side-channel attacks, and then we'll finally delve into the exquisite realm of attacks on public-key cryptography.

RELATED ARTICLES

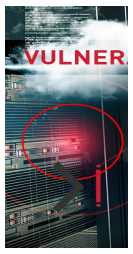

ıce



Zoom-Zoom: We Are Watching You (//research.checkpoint.com/2020/zoom-

Canadian banks targeted in a massive phishing campaign

Remote Cloud Execution – Critical Vulnerabilities in Azure Cloud Infrastructure (Part I)



Remote (Vulnerab Infrastru