Applied Cryptography and Network Security

Adam J. Lee
adamlee@cs.pitt.edu
6111 Sennott Square

Lecture #21: Subverting Cryptography
March 20, 2014

University of Pittsburgh

Announcements

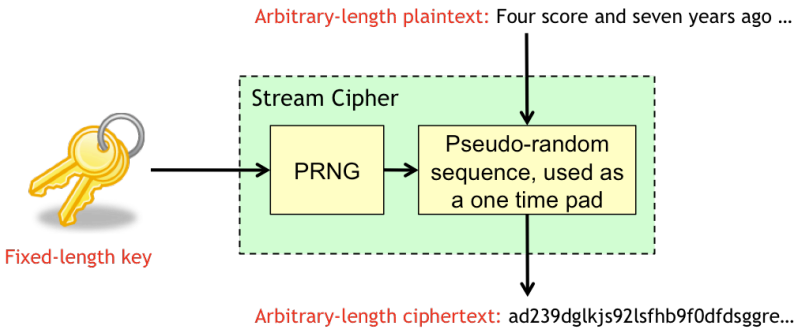
Project phase 4 is out!

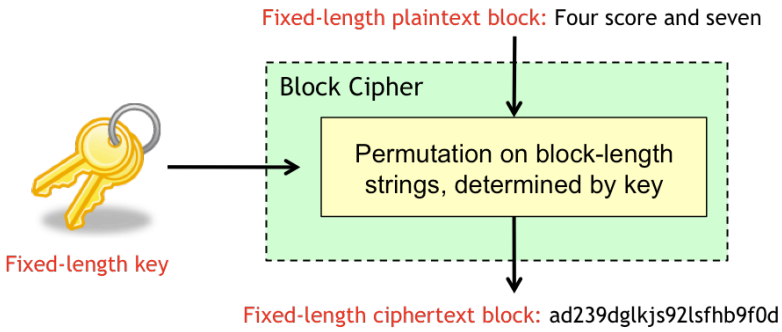
- Due: Thursday 4/3
- Defense discussion by 3/38

Interested in an MSIS or MST degree? Check out the SFS program at the iSchool:

- http://www.ischool.pitt.edu/resources/scholarship-for-service.php
- Deadline: March 30th, 2014

So far, we have treated cryptography largely as a black box





This certainly simplifies the engineering process

- Smart guys build the boxes, we can just put them together!
- Just worry about I/O, not the (messy) details

Unfortunately, it also abstracts away lots of details…

Today, we'll see how these types of details can cause the downfall of otherwise hardened black boxes

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- Chinese lotteries
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Can we break symmetric key cryptography without brute forcing the keyspace?

In theory, there is no difference between theory and practice. In practice, there is.

-- Yogi Berra

In theory, cryptographic keys are chosen randomly.

- For an m-bit keyspace, each of the 2^m keys is equally likely
- As such, figuring the key out requires a brute force search

In practice, things are not really random

- Computers are deterministic machines!
- Keys are picked using a pseudo-random process
- Figuring out this process can yield keys with less than brute-force effort

Question: How many people have used a PRNG? How did you seed it?

Goldberg and Wagner used this insight to break Netscape's key generation routines

All web browsers that use SSL need to generate secret keys that are exchanged during the connection setup process

- Typically, this is done using a cryptographically strong PRNG
- The initial seed to the PRNG determines the keys that are generated

After some reverse engineering, PRNG was found to be seeded using

- The current time
- The process ID of the web browser (PID)
- The parent process ID of the web browser (PPID)

The result: 128-bit keys were generated from 47 bits of randomness!

With a user account on the machine, the search could be reduced to about 10^6 = 1,000,000 guesses!

How can we prevent this?

What is one way that we might be able to prevent these types of well-known coding errors?

Case study: Kerberos v4

Kerberos is a popular network authentication protocol

- Initially developed at MIT as part of project Athena
- Used within the department for AFS "tokens"

Note: Kerberos is available as piece of widely used and widely studied open source software

The security of Kerberos depends on 56-bit DES or 112-bit TDES keys

In 1997, a group of grad students at Purdue showed that Kerberos v4 used the XOR of several 32-bit quantities to seed its PRNG

- Timing information
- PID and PPID
- Count of keys generated
- Host ID

So what does this tell us?

Observation: The XOR of any number of 32-bit quantities is 32 bits long!

- In 1997, brute forcing this 32-bit seed space only took about 28 hours!

As if that wasn't bad enough, many of these quantities are predictable

- Timing information can be guessed
- Process IDs can be read if the attacker has an account on the system
- The host ID is known
- …

The result of the above is that the actual entropy of the 56- or 112-bit keyspace is reduced to 20 bits!

- This can be brute forced in a few seconds…

The moral of the story: Not even open source software that has been scrutinized by lots of smart people is totally free of errors

An interesting aside…

This weakness was first discovered nearly a decade earlier (1988)

After this initial discovery, a stronger PRNG was written and checked into the Kerberos source tree, but was never actually used!

- The PRNG was written as a new function (not a replacement)
- Extensive use of `#define` statements in the code obscured which PRNG was actually being used

The result: Good code was available, but bad code was used…

The lesson? Don't get too cute and write incomprehensible code!

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- Chinese lotteries
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Surely, breaking *some* systems must require a brute force search of their keyspace, right?

If all else fails, any cryptosystem can be broken by "simply" to decrypt a ciphertext with all possible keys

Recall: NSA reduced Lucifer's keyspace when developing DES

- Lucifer was a 64-bit cipher, DES is a 56-bit cipher
- Many cryptographers began to speculate as to why this was done

Certainly, brute forcing a 56-bit space on a single computer would take a very long time, but why use just a single computer?

Quisquater and Desmendt discussed the possibility of a Chinese lottery

- Key-cracking hardware could be built into TVs and radios
- State-run media could farm out searches to these devices
- A back of the envelope example:
  - Keys tested at 1 million/second
  - 1 Billion TVs/radios
  - Break a DES key in about a minute…

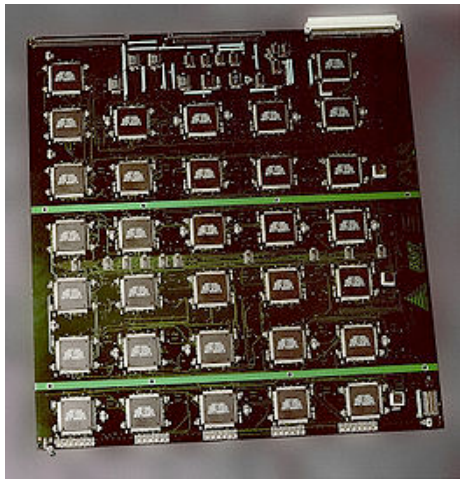
Deep crack is a slight less "big brother" version of this theory

The Electronic Frontier Foundation (EFF) is a digital rights advocacy group based in the US

To demonstrate that 56-bit keys are not secure enough for widespread use (despite the government's claims to the contrary), the EFF built a machine called Deep Crack

Deep Crack is an extremely parallel machine (esp. by 1997 standards)

- 29 individual circuit boards
- 64 specially-designed DES-cracking chips per board
- 1,856 chips running in parallel could test about 90 billion keys per second!
- The entire DES key space could be tested in about 9 hours



With a price tag of about $250,000, Deep Crack was well within the budget of many criminal organizations and/or enemy goverments

Take home message

Definitions of words like "big", "infeasible", and "difficult" are rooted in hardware, software, or algorithmic assumptions. Make sure to revisit these assumptions often!

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- Chinese lotteries
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Is it possible to subvert public key systems without attacking the mathematics?

Question: Say that Alice runs a digital notary. We want Alice to sign the message M for us, but do not want her to see M.

Surprisingly, we can trick Alice into doing this!

- Let (n, e) be Alice's public key
- Let d be Alice's private key
- Let X be a random number that we pick
- Let Y = X^e mod n
- Let M_D = YM mod n be a "decoy message"

M_D is a randomized version of M

Now, we get Alice to sign M_D, returning U = M_D^d mod n

Note: UX^{-1} mod n = $M_D^d X^{-1}$ mod n // $U = M_D^d$

= $Y^d M^d X^{-1}$ mod n // $M_D^d = (YM)^d = Y^d M^d$

= $X^{ed} M^d X^{-1}$ mod n // $Y = X^e$

= $X M^d X^{-1}$ mod n // $X^{ed} = X$

= M^d mod n // $XX^{-1} = 1$

Alice signed M without knowing it!

If Alice also decrypts using her signing key, we can read her private messages!

How? It's easy! Let:

- $C = M^e \bmod n$ be a message encrypted to Alice
- R be a random number
- $X = R^e \bmod n$
- $Y = XC \bmod n$

Now, we ask Alice to sign Y, which gives us $U = Y^d \bmod n$

Note:
$$
\begin{aligned}
R^{-1}U \bmod n &= R^{-1}Y^d \bmod n && // \; U = Y^d \bmod n \\
&= R^{-1}X^dC^d \bmod n && // \; Y = XC \bmod n \\
&= R^{-1}R^{ed}C^d \bmod n && // \; X = R^e \bmod n \\
&= R^{-1}RC^d \bmod n && // \; R^{ed} \bmod n = R \bmod n \\
&= M^{ed} \bmod n && // \; RR^{-1} = 1,\ C = M^e \bmod n \\
&= M \bmod n && // \; M^{ed} \bmod n = M \bmod n
\end{aligned}
$$

This is probably not a good thing…

Why do these attacks work?!

RSA has what is known as a multiplicative homomorphism

- $(X^z \bmod n)(Y^z \bmod n) = XY^z \bmod n$
- I.e., $E(x)E(y) = E(xy)$ if E is the RSA encryption function

The attacks that we just talked about used RSA to operate on raw data

- Data is represented as plain old numbers
- Each number encrypted directly

In real life, RSA is not used this way! Padding functions like OAEP are used to randomly pad message prior to encryption or signing

- I.e., M becomes P(M)
- $P^{-1}(P(M)) = M$
- $P^{-1}(D(E(P(M)))) = P^{-1}(P(M)) = M$

Note: Given $E(P(M_1))$ and $E(P(M_2))$, we can calculate $E(P(M_1)P(M_2))$. However, $P^{-1}(D(E(P(M_1)P(M_2)))) = P^{-1}(P(M_1)P(M_2)) \neq M_1M_2$.

Take home message

Breaking public cryptography does not need to involve "breaking" mathematics. More often than not, misusing a protocol or two can be just as effective!

Topic Outline

Brute force attacks without using brute force

- SSL key generation
- Kerberos v4

Brute force attacks against symmetric key ciphers

- Chinese lotteries
- Deep Crack

Subverting public key cryptography protocols

Attacking real world implementations

- Timing analysis of RSA attacks
- Power analysis of DES
- Keyjacking cryptographic APIs

Remember that whole "successive squaring" trick from a few lectures ago?

```
// Goal:  Calculate m^d mod n
int pow(m, d, n)
    bitvector b[] = binary representation of d
    int r = 1
    for(int i=0; i <= b.length(); i++)
        r = r*r mod n
        if(b[i] == 1) then r = r * m
    return r
```

MSB in b[0]

Note that $5_{10} = 101_2$

Example: Computing pow(2, 5, 64)

Iteration	r
0	1
1	$1^2 \times 2 = 2$
2	$2^2 = 4$
3	$4^2 \times 2 = 32$

Observation: This algorithm does more work when bits are set to 1

We can leverage this observation to launch an attack on implementations of RSA!

In 1995, Paul Kocher described and demonstrated a timing attack against square-and-multiply implementations of RSA

The gist: If we know C, we don't know d, and we can measure the time that the operation C^d takes, we can eventually learn d by observing many such operations

So, why does this work?

- We can make a guess for the first bit of d
- If our guess is correct, we can predict
  - The intermediate result r
  - How long the algorithm will take
- If our guess is incorrect, we don't learn anything

That only explains one round, how does this work for a large key?

Details, details, details…

Consider the following observations, assuming the a w-bit key:

- The time a decryption operation takes is the sum of the individual times for each of the w bits
- If we can correctly guess the first k bits of the key
  - We can calculate the time required for the first k operations
  - The remaining terms in the sum are essentially random
- Over a large enough sample of C, the difference between how long we calculate C^d to take versus how long C^d actually takes is a distribution whose variance is proportional to k

In short, by guessing correctly, we can reduce the variance of this distribution and learn k!

Furthermore, this method allows us to learn the private key in O(w) time, rather than $O(2^w)$ time

This seems really esoteric…

Sure, I'm convinced that this is possible, but how useful is this attack likely to be in practice?

In 2003, Brumley and Boneh showed how to successfully launch timing attacks against OpenSSL-based Apache web server over the Internet!

The lesson? Over time, attacks only get better!

If timing information is useful, what can we do with other sources of information?

Obvious statement: Software runs on hardware, hardware requires power

- Algorithms leak information via the power that they consume
- By monitoring the power consumption of hardware, we may be able to learn something about the inputs to the algorithms running
- Keys are important inputs to cryptographic algorithms!

Kocher et al. demonstrated a viable attack for learning the DES keys used by certain types of smart cards

Their attack was very clever

- Recall that DES keys are 56 bits long
- They observed an odd series of 56 spikes in the power trace
  - Why? Software checking the parity of the DES key
  - Spikes were at one of two heights, corresponding to 1 or 0 in the key
- By watching one encryption or decryption, they learned the key!

Even a good implementation and solid hardware can be subverted by a bad API…

Most often, cryptographic routines are not built directly into applications, but rather exist as a library that is used by applications

So-called keyjacking attacks work as follows:

1. Run a malicious executable on the viticim's machine
2. Use this executable to begin intercepting calls to the victim's crypto API

This allows the attacker to do all kinds of nasty stuff

- Export secret/private keys to other applications
- Decrypt private messages
- Forge signatures
- …

These attacks are very hard to depend against, as once the system is compromised (e.g., via a buffer overflow exploit), there is nothing much that can be done!

Take home message

Cryptographic hardware exists where people can observe it. Cryptographic software may leak information about the inputs that it uses. Our nice little black boxes need to exist in the real world, and are thus subject to scrutiny and attack.

Conclusions

Treating cryptography as a black block simplifies our lives as engineers, but gives us a reason to ignore certain types of threats

Today we say that

- Sometimes keys can be guessed without a brute force attack
- Sometimes brute force attacks are tractable by thinking outside of the box
- Naive use of public key cryptography can lead to undesirable outcomes
- Engineering decisions when building cryptosystems can be used to break these cryptosystem when they are deployed

As a result, it is important to keep up to date on how to safely use modern cryptography

Next time: ACLs and Capabilities