Lab 3: Breaking Bad Crypto

In this lab, we first get some experience encrypting and decrypting messages using an off-the-shelf crypto API. Then we use our new understanding of cryptographic tools to break some badly-designed applications.

Due: 10pm Thursday, Apr 24, 2014

Preliminaries

Setting up SSH on your VM

Jared Evans has a nice tutorial on setting up an Ubuntu VM for console access (http://www.jaredlog.com/?p=1484).

The short version is that your VM needs to know that it has a (virtual) serial port interface, and it needs to run a login process on that interface. The tutorial linked above shows you how to do this.

You'll probably need to reboot your VM. Then you can access your VM over SSH like this:

ssh -t vm@seclabXX.cs.pdx.edu console

(Replace seclabXX with the actual name of your VM host. Leave the username (vm@) just as it is -- this is a special account that exists to provide SSH access to our VM's.)

If you don't see a login prompt, press the 'Enter' key, and the prompt should appear.

You can also start, shutdown (nicely), reboot, or destroy (hard shutdown) your VM in the same way. Simply replace the command "console" in the above SSH invocation with the command you want to perform. For example, to reboot your vm, you would do:

ssh -t vm@seclabXX.cs.pdx.edu reboot

And that's it. Happy hunting!

Creating your "inside" SSH keys for Git

Very soon, we will have a Git server that can be accessed from both inside and outside the virtual lab network. For access to Git from outside the virtual network, you already have SSH keys on your machines, and you turned in your public keys through D2L. For access to git from *inside* the lab network -- that is, from your VM -- you also need a public and private key on your VM.

To create your "inside" keys, open a terminal on your VM, and run

ssh-keygen -t rsa

and save the resulting public key to **/home/user/.ssh/id_rsa.pub**.

And that's all you need to do. A script on the hypervisor will come along to collect your public key sometime this evening.

Challenge Problems

Problem 1: Encoding Binary Data (Part 1)

Connect to the server at 192.168.14.10 on port 3001 and request the flag as in Lab 02 (lab02.html).

Unlike in Lab 02, where the flags were always ASCII text, this flag will be sent to you as raw binary bytes. The format of the server's response message will be "FLAG *length flag*", where *length* is the length of the flag in bytes, written as a standard decimal integer.

Encode the flag as ASCII characters using **hexadecimal** encoding with Python's binascii (https://docs.python.org/2/library/binascii.html) module or a similar facility.

Problem 2: Encoding Binary Data (Part 2)

Connect to the server at 192.168.14.10 on port 3002 and request the flag as in Lab 02 (lab02.html).

Unlike in Lab 02, where the flags were always ASCII text, this flag will be sent to you as raw binary bytes. The format of the server's response message will be "FLAG *length flag*", where *length* is the length of the flag in bytes, written as a standard decimal integer.

Encode the flag as ASCII characters using **base64** encoding with Python's base64 (https://docs.python.org/2/library/base64.html) module or a similar facility.

Problem 3: Simple Decryption (ECB)

Connect to the server at 192.168.14.10 on port 3003 and request the flag as in previous problems.

The server's response will be encrypted with AES in ECB mode and transmitted as raw bytes. The key, as a hexadecimal string, is 20 14 03 03 20 14 03 03 20 14 03 03 20 14 03 03. Use binasii to convert these hex digits to **16 raw bytes** in order to use them as an AES-128 key.

Decrypt the message and recover the flag.

Hint: PyCrypto provides a nice Python interface (https://www.dlitz.net/software/pycrypto/api/current/) to the relevant AES routines -- see Crypto.Cipher.AES

Problem 4: Simple Decryption (CBC)

Connect to the server at 192.168.14.10 on port 3004 and request the flag as in previous problems.

The server's response will be encrypted with AES in CBC mode and transmitted as raw bytes. The key, as a hexadecimal string, is 20 14 03 04 20 14 03 04 20 14 03 04 20 14 03 04. Use binasii to convert these hex digits to **16 raw bytes** in order to use them as an AES-128 key.

Decrypt the message and recover the flag.

Hint: In CBC mode, the first block of data is the initialization vector (IV), not ciphertext!

Problem 5: Verifying a Hash

Connect to the server at 192.168.14.10 on port 3005 and request the flag as in previous problems.

The server's response will be provided in plain text, but it will include a hash to let you verify the integrity of the message. The message format for the server's response is: "FLAG *flag hash*", where *hash* = H("FLAG *flag*"). The hash is computed using SHA-256 and is encoded in the message as hexadecimal ASCII characters.

Note: This server often returns garbled flags and/or incorrect hashes. Be sure to verify that hash!

Problem 6: Verifying a MAC

Connect to the server at 192.168.14.10 on port 3006 and request the flag as in previous problems.

The server's response will be provided in plain text, but it will include a message authentication code to let you verify the integrity of the message. The message format for the server's response is: "FLAG *flag mac*", where *mac* = MAC("FLAG *flag*"). The MAC is computed using HMAC-SHA1 and is encoded in the message as hexadecimal ASCII characters.

The key, as a hexadecimal string, is 20 14 03 06 20 14 03 06 20 14 03 06 20 14 03 06 20 14 03 06.

Note: This server often returns garbled flags and/or incorrect MACs. Be sure to verify that MAC!

Problem 7: Bad Encryption Password

Connect to the server at 192.168.14.10 on port 3007 and request the flag as in previous problems.

The server's response will be of the form "FLAG *flag*", encrypted with AES-128 in CBC mode and transmitted as raw bytes. The encryption key was generated by taking the MD5 hash of a dictionary word.

Find the password that generates the encryption key and decrypt the server's response to extract the flag.

Hint: Your VM has a list of several thousand dictionary words in the file /usr/share/dict/words. The password was typed in all lowercase.

Problem 8: Network Proxy

There is a server at 192.168.14.10 on port 3008. Some clients in the network are unable to connect to this address themselves, so they must use your VM as a proxy.

Listen on port 3808 for connections from the clients. When a client connects to your proxy, you should make a new connection to the server and relay messages back and forth between the client and server.

There are two flags for this problem: one from the server and one from the client. You can extract them from the messages passed back and forth, but to obtain the flags, you must pass each message accurately and without too much delay.

Problem 9: MAC Length Extension Attack

There is a server at 192.168.14.10 on port 3009. Some clients in the network are unable to connect to this address themselves, so they must use your VM as a proxy.

Listen on port 3909 for connections from the clients. When a client connects to your proxy, you should make a new connection to the server and relay messages back and forth between the client and server.

The clients will request the values of several items from the server. The clients' requests will have the form "GET *item1 item2 ... itemN mac*", where the MAC is a secret-prefix MAC computed using an unknown, 64-bit secret value *secret* and an MD5 hash, like this: *mac* = MD5(*secret*||"GET *item1 item2 ... itemN*"). The server will only provide the items' values if the MAC is correct.

One of the items on the server is "FLAG", but the client will never request it on its own. Your task is to perform a length extension attack on MD5 to add "FLAG" to the list of requested items. See this blog post (https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks) for an in-depth description of the attack. (Fortunately for you, the request parser in the server is very liberal in what it accepts as a valid request.)

Note that this exercise is very similar to a real vulnerability in Flickr (http://netifera.com/research/flickr_api_signature_forgery.pdf) in 2009. See the linked PDF for another description of the vulnerability and the attack technique.

Problem 10: Bit Flipping Attack

There is a server at 192.168.14.10 on port 3010. Some clients in the network are unable to connect to this address themselves, so they must use your VM as a proxy.

Listen on port 3110 for connections from the clients. When a client connects to your proxy, you should make a new connection to the server and relay messages back and forth between the client and server.

The clients' requests are encrypted using AES-128 in OFB mode using an unknown secret key. The request plaintexts have the form "GET *item1 item2 ... itemN*". The server only provides the requested values if the encrypted message decrypts to a valid request. Server responses are not encrypted.

Your task is to modify a client request so that the server provides the value of FLAG in its response. Fortunately for you, we have reports that the clients always request the same items in the same order. These are: DOG, FROG, GRIP, STOP, and FLY.

~~Problem 11: Diffie-Hellman Key Exchange~~ POSTPONED Until Lab 4

There is a server at 192.168.14.10 on port 3011. To retrieve its flag, you must first perform a Diffie-Hellman key exchange with the server to derive a shared secret key. You can then use the shared secret key to send an encrypted request ("GET FLAG") and decrypt the server's encrypted response (the plaintext will be "FLAG *flag*").

For a review of the Diffie-Hellman key exchange, see Mark Loiseau's tutorial (http://blog.markloiseau.com/2013/01/diffie-hellman-tutorial-in-python/). Here, our server uses the prime p = 999,959 and the generator g = 2.

Note: In order to make this exercise possible without importing additional libraries, we're using small numbers that are **NOT** safe for use in the real world. (Using g = 2 is OK. In practice, p should be **much** larger.)

The protocol works as follows:

1. When you connect, the server immediately sends you its public key: "PUBKEY *pubkey*", where *pubkey* is a standard decimal-encoded integer. If the server's private key is *s*, then the server's public key is equal to g^s mod p.

2. You should reply with your public key in the same format: "PUBKEY *pubkey*". If your private key is *c*, then you can compute your public key as g^c mod p.

3. Both sides then derive the session key as MD5(g^(sc) mod p).

4. You can now encrypt your request ("GET FLAG") with AES-128 in CBC mode using the session key. You send your encrypted request to the server as raw binary bytes.

5. The server attempts to decrypt your request, and if successful, it sends back a response ("FLAG *flag*") encrypted with the same key in the same way.

Question: What's wrong with this scheme? How many ways can you find to attack it?

~~Problem 12: Diffie-Hellman MITM Attack~~ POSTPONED Until Lab 4

There is a server at 192.168.14.10 on port 3012. To retrieve its flag, clients must first perform a Diffie-Hellman key exchange with the server to derive a shared secret key. The client can then use the shared secret key to send an encrypted request ("GET FLAG *cookie*") and decrypt the server's encrypted response (the plaintext will be "FLAG *flag*").

Note that this server only provides the flag if a valid cookie is included in the request.

Some clients cannot connect to this server directly, so they will attempt to connect to your IP address on port 3212. Your job is to proxy their connections to the server and perform a man-in-the-middle attack to extract the flag.