Analysis of Authenticated Encryption, Block Cipher Implementations, and Attacks

Nicholas Gaunt, Bill Hass, Myles Pollie, Robert Minnema

December 14, 2016

1 Introduction

Symmetric key cryptography involves two parties, Alice and Bob, who have somehow obtained the same symmetric key, K, and wish to communicate with secrecy, integrity, or both over a public channel. An encryption scheme can be used to achieve secrecy and a message authentication scheme can be used to achieve integrity. Today's most widely used encryption and message authentication schemes are built upon cryptographic primitives such as pseudorandom generators (PRGs), pseudorandom functions (PRFs), and hash functions. Block-ciphers are one such set of cryptographic schemes built upon those primitives to provide security guarantees. They are an important tool for cryptographers and security engineers because they have several favorable properties: faster and more efficient than asymmetric schemes, relatively simple constructions, some are parallelizable, and are quantum-resistant.

In this paper, we analyze several popular block-cipher constructions that are built upon PRFs and designed to achieve either secrecy or integrity, and we demonstrate how common block-ciphers can be attacked by an adversary. To carry out our investigation, we built our own implementations of encryption, message authentication, and authenticated encryption (AE) schemes in C++ using OpenSSL's libcrypto for the Advanced Encryption Standard (AES) PRF. Then, we developed practical attacks based on theory learned in class to decrypt AES CBC-mode and CTR-mode ciphers, forge message authentication codes (MACs) in CBC-MAC, and attack both Encrypt-and-Mac and Mac-then-Encrypt AE schemes.

1.1 Goals

1. Deepen understanding of block-cipher mechanics and authenticated encryption schemes.
2. Understand how theoretical failures of block-ciphers can lead to failures in practice.
3. Develop and implement practical attacks and come up with defenses.
4. Hands on experience implementing secure symmetric key systems using a popular, open source, cryptographic library : OpenSSL.

1.2 Background

Our code has been compiled with g++ using –std=c++11 and runs on CAEN computers in a Linux environment. The source can be found in the same folder as the assignment submission called:

gaunt_hass_pollie_minnema_eecs475.tar.gz

Because the latest versions of OpenSSL [6] have fixed the vulnerabilities we present in our paper, we statically compiled libcrypto from v1.0.0-beta4 [4] for use in our implementations. The keys we used are hard-coded into our software.

In several experiments we use a client-server architecture with sockets. A simple packet structure protocol is used to facilitate communication and is as follows: '$< header >$ $< data >$'. The $< header >$ and $< data >$ are separated by a single whitespace, and the $< header >$ consists of three whitespace separated

fields: '$< type >$ $< enc/dec >$ $< length >$'. $< type >$ can be one of CBC, CTR, TAG, EAT, TTE which stand for the various types of schemes supported by the server. $< enc/dec >$ can be either ENC or DEC to indicate which direction the server should run the requested scheme. $< length >$ is an integer between 1 and 1024 to indicate how many bytes are in the $< data >$ payload. $< data >$ is an array of unsigned chars (raw bytes).

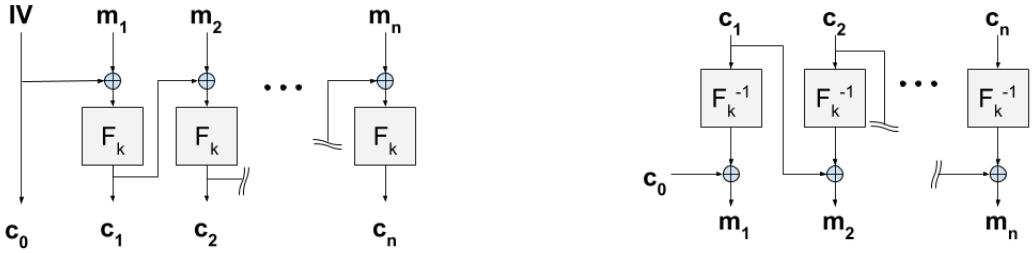
In the rest of the paper, we use the notation from class and from Katz and Lindell [2]. Namely, $k \in \{0,1\}^n$ denotes an n-bit, uniformly random variable k; $m_1 \| m_2$ denotes concatenation of two variables $m_1 \in \{0,1\}^n$ and $m_2 \in \{0,1\}^n$; and $F_k^{-1}(y)$ denotes the inverse of a keyed function $F_k(x)$.

2 Encryption Schemes

We have analyzed two common block-cipher encryption schemes: Cipher Block Chaining (CBC) and Counter (CTR) modes using 128-bit AES as the PRF.

2.1 AES-CBC

AES-CBC is a block-cipher that requires a message to be padded so that its length is a multiple of the block size. A diagram of the CBC-mode block-cipher is shown in Figure 1. Encryption works by initially padding m_1 with IV, then chaining the resulting output from F_k to be padded with the next block of the message. Decryption requires that the PRF is invertible, and works by XORing the first block of the cipher (the IV) with the output of F_k^{-1}.



(a) Encryption

(b) Decryption

Figure 1: Cipher Block Chaining (CBC) Mode Block Cipher

2.1.1 Security Analysis

The fact that decryption has the form $m_1 = c_0 \oplus F_k^{-1}(c_1)$ means that if an adversary modifies a bit in the ciphertext, the corresponding bit is modified in the plaintext message. Using this, an attacker can induce a padding error during decryption. Then, given an oracle that returns an error for decryption failure, in practice this is typically a server, an attacker who has intercepted a ciphertext c can modify c in a way that exploits this padding and forces the oracle to leak information about the plaintext. This will allow the attacker to recover the entire plaintext. Our implementation performs AES 128-bit CBC encryption which uses a blocksize of 16 bytes. The method of padding that was used is PKCS7 which pads the final message block with with b blocks of $0xb$. For example if the final block of a message contained 15 bytes, it would be padded with 1 byte of $0x1$, a 14 byte message would be padded with 2 bytes of $0x2$, etc. A message that is a multiple of the blocksize will have a full block of $0x10$'s appended to the message. An example of a 2 block message being padded before encryption is shown below in Figure 2.

2.1.2 Attack Implementation

Here we describe how we implemented our padding oracle attack. It works in two stages: (1) The padding is determined in the last message block; (2) Starting with the last message block and working towards the

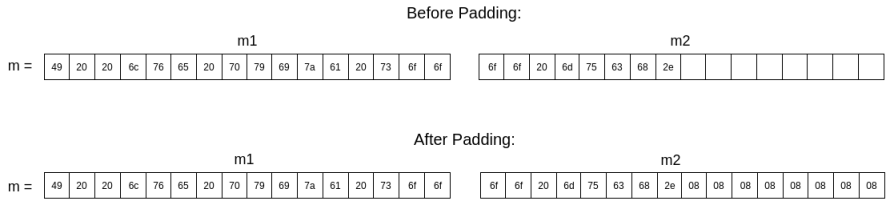
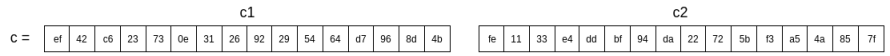


Figure 2: PKCS7 Padding on a 2 block message

first, bytes are determined 1-by-1 until the full plaintext is discovered. Say an attacker has intercepted the following ciphertext c:



We can discover m_2 by modifying the previous ciphertext block c_1. To find where the padding begins we modify c_1, byte-by-byte from left to right, until the oracle returns a padding error. The idea is that if the last message block was all padding, we would get a padding error right away because the padding wouldn't be consistent in m_2 (all $0x10$'s), otherwise if the last message block is not all padding we wouldn't get a padding error because decryption treats the message bytes opaquely - if it isn't padding it doesn't care. We begin by modifying the first byte of c_1 and sending the resulting ciphertexts to the padding oracle. If it were to fail, we would know that all of m_2 is padding. In our example, it doesn't fail and we continue modifying bytes of c_1. We find that the padding begins at index 7 (indexes begin at 0). Let *paddingstart* represent this index. So we only care about the first 7 bytes of the encoded plaintext, as the last 9 are filled with $0x9$.

To begin cracking the plaintext bytes we care about, we construct two vectors, $v1$ and $v2$ in the following general form where $b = blocksize - paddingstart$:

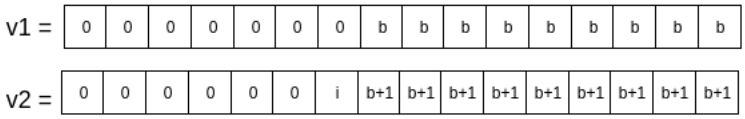


Figure 3: Padding Oracle Attack: General Form Vectors

In our case, padding start was found to be 7, therefore $b = 16 - 7 = 9$. Then, we have the two vectors $v1$ and $v2$, shown on the next page in Figure 4.

Let our guess be defined as $m_2 = \{..., 0xB, 0xb, ..., 0xb\}$ where 0xB is the plaintext value we are trying to guess. By XORing c_1 with the first vector, $v1$, we notice that upon decryption $c_1 \oplus v1$ is XORd with $c_1 \oplus F_k^{-1}(c_2) = v1 \oplus F_k^{-1}(c_2)$ This causes the last b bytes of m_2 to go to 0. Then, when we further XOR $v2$, the padding becomes $b + 1 = 0xa$ and our guess has the form $0xB \oplus i$. Now we are free to try all 255 values of i until we find one that causes the resulting decrypted block to end in 10 bytes of $0xa$. If we try a value of i and the oracle returns a padding error, increase i by 1, reconstruct $v2$, and send the new $v1 \oplus v2 \oplus c_1$. If the oracle returns no padding error, we have now found the last value of the plaintext because it must be the case that $0xa = 0xB \oplus i$ in order for the padding to be correct. We continue the attack with our updated ciphertext, increase the padding $0xb$ by 1, and a new guess 0xB for the next plaintext byte. This gives us our next set of vectors $v1$ and $v2$, shown on the next page in Figure 5.

We then continue the attack doing the above steps until all values of m_2 are found. To extend the attack to crack m_1, notice that the adversary can simply choose not to send c_2. In that case, the decryption will look at the last byte of m_1 to check for padding. Construct $v1$ and $v2$ as before in the general case, but instead let $0xb = 1$ and $v2 = \{0, ...0, i\}$. The goal is to find an i that makes $0xB \oplus i = 0x1$. Since there exists such an i, and the attack can be extended down to the first message block, we can recover the entire plaintext. To run this attack:

```
$> ./oracle
```

Source code can be found in *src/oracle.cpp*.

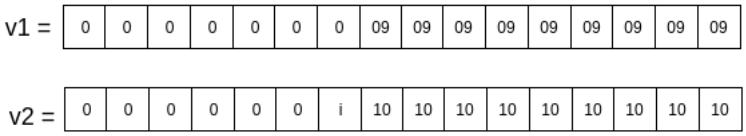


Figure 4: Padding Oracle Attack: Example Vectors

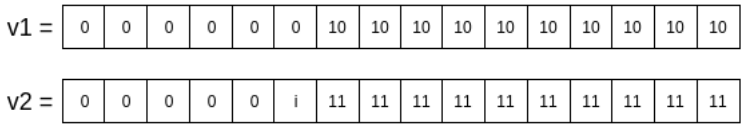


Figure 5: Padding Oracle Attack: Example Vectors - 2nd Iteration

2.1.3 Discussion

The padding oracle attack we demonstrated is inherently enabled by the structure of the CBC-mode block cipher. When using 128-bit AES CBC-mode encryption, an attacker only needs to submit a maximum of 256 queries to the oracle to learn each byte of plaintext. Thus, the complexity of the padding oracle attack is linear - $O(n/8 * 256) = \text{poly}(n)$ - and is therefore a feasible attack. There are a couple ways to defend against padding oracle attacks. One can use authenticated encryption by applying an "encrypt-then-tag" construction to CBC-mode which prevents CCA. Another method is to not return a specific padding error when padding is malformed, rather pass the message to the application layer where a generic error message can be thrown.

2.2 AES-CTR

AES-CTR mode works by taking a randomized initialization vector, passing it as input to a PRF, and using that as a pad to XOR with a message block. Once one block is encrypted, the initialization vector is incremented and the process is repeated until the entire message is encrypted. Figure 7 shows the construction for the CTR-mode block cipher.

(a) Encryption (b) Decryption

Figure 6: Counter (CTR) Mode Block Cipher

2.2.1 Security Analysis

CTR mode blockciphers are not CCA secure. Just by analyzing the construction, we can see that for decryption $m_1 = c_1 \oplus F_k(IV + 1)$, and if an adversary manipulates a bit in c_1, then the corresponding bit in m_1 will be flipped. This means an active adversary has the ability to meaningfully change a message received by Bob if they modify the ciphertext in-transit.

CTR mode is CPA secure as an encryption scheme. The incrementing counter added to the randomized IV guarantees distinct input to the PRF which means the output behaves pseudorandomly. This is XORed with a block of the message gives pseudorandom looking output.

However, if an IV_a is reused or in the incrementing of a subsequently used IV_b we end up with some $IV_b + t = IV_a$ we end up with an overlapping window of used pads. From this we can take the two resulting

ciphertexts and XOR them:$c_1 \oplus c_2 = m_1 \oplus IV_i \oplus m_2 \oplus IV_i = m_1 \oplus m_2$. With some knowledge of the message structure and adversary can easily decrypt both messages.

One of the key points to note about this is that once a pair of message blocks have been recovered, one can obtain the pad used for that block simply by XORing the obtained message with the corresponding ciphertext block. Once this pad is obtained, any message block that reuses this IV_i and the same key can be decoded instantly because the pad will once again be the same.

2.2.2 Attack Implementation

We implemented an attack on CTR-mode for a scenario where there was a reused IV or equivalently a window of reused IV increments. The attack reads the two ciphertexts or ciphertext chunks and XORs them. The implemnted attack makes the following assumptions: The character set is restricted to printable ascii characters and the dictionary of possible words is that of English.

A guess is entered in by the user. This guess is then XORed at all possible locations in the XOR of the ciphertexts. For each string created like this, a function looks for impossible patterns in punctuation and alphanumeric characters. The ones that pass are returned to the user with their locations. In this way, if a string guessed by the attacker is contained anywhere in either message it will produce the corresponding characters from the other message. In this way, the attacker can progressively extend their guess to eventually the whole message.

The attack can be ran by:

```
$> ./ctr_attack <file1> <file2>
```

Where $< file1 >$ and $< file2 >$ are the files containing the raw cipher text. The source can be found in *src/CTRModeReusedIVAttack.cpp*.

```
" the "  Initial guess
"e oth"  Returned potential string

"e other "   New guess from expanding
" the tim"   Returned potential string

" the time " New guess from expanding
"e other da" Returned potential string

...
```

Figure 7: Progressive Message Decryption

2.2.3 Discussion

Considering the feasibility for this attack is a somewhat complicated question. It depends on the information the attacker has about what the message content could be. If we consider for a moment what the situation would be like if we had no assumptions about what the content of the message could be. Any combination of characters could be a possible message so there's no way of knowing if our guess is correct. This nullifies the attack completely. However this is almost always not the case.

The assumptions we have listed above serve to rule out not only possible guesses but also guesses where the message counterpart is in violation of these assumptions. Operating under these assumptions we can shrink the search space of possible n-byte messages.

An interesting property of this method is that the longer the ciphertexts are, the more likely a given word is contained in it. For example searching for the string " the " is likely to come up multiple times in a longer message. For each "hit" on a guess, you have more opportunities to expand your guess string at a given location.

3 Message Authentication Schemes

3.1 CBC-MAC

CBC-MAC is a method of using the Cipher Block Chaining mode of operation as a tagging algorithm for the purpose of authentication. The key differences between CBC-MAC and CBC mode encryption are that the IV for CBC-MAC is fixed to all zeroes, that the tag from CBC-MAC is formed from the output of the last block in the chain, and that the pseudorandom function used need not be invertible. For this attack we use 128-bit AES as the underlying PRF for CBC-MAC. Our server is configured to compute and return the tag for a given message on request (in order to act as a tagging oracle), and also to verify tagged messages.

Verification is done in the canonical way, by recomputing the tag from the received message and comparing it to the received tag, as described in Section 3.2.

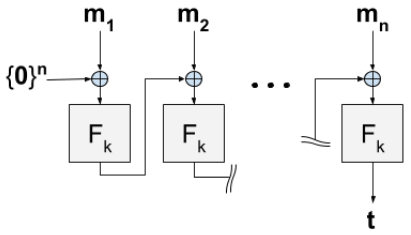


Figure 8: Cipher Block Chaining (CBC) MAC

3.1.1 Security Analysis

When restricted to fixed-length inputs (by having the verifier reject messages which are too long or too short), CBC-MAC is proven to be unforgeable under chosen message attack (UF-CMA), assuming the PRF used is secure. It is also possible to have a variable length CBC-MAC construction which is UF-CMA secure by running all inputs through a prefix-free encoding scheme, which ensures that no message (after encoding) is a prefix of any other message after encoding. When used with prefix-free encoding, the CBC-MAC becomes a pseudorandom function, which is trivially UF-CMA secure.

However, when CBC-MAC is used with variable length messages and no prefix-free encoding is used, the construction becomes vulnerable to length extension attacks that allow an adversary to forge the tag of any arbitrary message without having the private key.

3.1.2 Attack Implementation

We implemented the CBC-MAC scheme using 128-bit AES-CBC with an IV of 0^{128} where only the last block of the ciphertext was returned as the tag. Our server did not check the length of the message before tagging, but instead passed the message directly to the AES-CBC routine which can be an easy mistake to make as an implementer of CBC-MAC. In general, our forgery works by first obtaining tags for two messages. Then we combine the two messages and use the knowledge of the two tags to produce a forgery.

Two messages m_1 and m_2 are generated, where $|m_1| = |m_2|$. For simplicity, we will assume both m_1 and m_2 are one block size each, i.e. 128 bits. We query our server on m_1 to get $t_1 = \mathsf{Tag}_k(m_1)$, and on m_2 to get $t_2 = \mathsf{Tag}_k(m_2)$.

Because AES-CBC employs PKCS#7 padding, the input to CBC-MAC for each of m_1 and m_2 is actually two blocks for each, the second block containing 16 bytes of $0x10$. Therefore, $t_1 = \mathsf{Tag}_k(m_1 \| (0x10)^{16}) = F_k((0x10)^{16} \oplus F_k(m_1 \oplus 0^{128}))$ and $t_2 = \mathsf{Tag}_k(m_2 \| (0x10)^{16}) = F_k((0x10)^{16} \oplus F_k(m_2 \oplus 0^{128}))$

We can XOR m_2 with t_1 and then append it to m_1 in order to create a new message which results in a tag value that we already have. Since we need to take padding into consideration, our message incorporates the padding that was added to m_1 before computing AES-CBC. Our new message is $m^* = m_1 \| (0x10)^{16} \| m_2 \oplus t_1$. Because padding will be added to make this three-block message into four blocks, the tag of this message is $t^* = \mathsf{Tag}_k(m^* \| (0x10)^{16}) = F_k((0x10)^{16} \oplus F_k(m_2 \oplus t_1 \oplus F_k((0x10)^{16} \oplus F_k(m_1 \oplus 0^{128})))) = F_k((0x10)^{16} \oplus F_k(m_2 \oplus t_1 \oplus t_1)) = F_k((0x10)^{16} \oplus F_k(m_2 \oplus 0^{128})) = t_2$.

XORing m_2 with t_1 cancels out when tagging m^* such that $\mathsf{Tag}_k(m_1 \| (0x10)^{16} \| m_2 \oplus t_1 \| (0x10)^{16}) = \mathsf{Tag}_k(m_2 \| (0x10)^{16})$. Therefore, $t^* = t_2$ and we can send t_2 as our tag for m^*.

3.1.3 Discussion

One simple method of avoiding the above attack is to simply prepend each message with its length (in bits). This is an easy form of prefix-free encoding, which turns CBC-MAC into a PRF, making it unforgeable. However, one disadvantage of this scheme is that it requires that the message length is known in advance, prior to computing the MAC.

Other attacks against CBC-MAC become possible if implemented improperly. One example is when CBC-MAC is used as part of an "encrypt and tag" authenticated encryption scheme where the mode of encryption is also CBC and the same key is used for both CBC-MAC and CBC encryption. In that case, an adversary is able to modify any part of the ciphertext except for the last block. When decrypted this produces a plaintext very different from the original, but which still produces the same tag when run through CBC-MAC. Another example is when a random IV is used as input to the CBC-MAC block-cipher instead of all 0's. If the IV is controllable by an adversary, they can modify bits in the IV as well as corresponding bits in the message to produce the same tag.

Finally, it's important to note that authentication schemes are all susceptible to replay attacks because the security constraints do not consider messages that are replayed. It's up to the implementation to provide a source of *freshness* to mitigate replay attacks.
To run, first start the server:

```
$> ./server
```

Then, in a new terminal, start the client using the localhost interface against EAT:

```
$> ./mac_attack localhost
```

Source code can be found in *src/mac_attack.cpp*.

3.2 Canonical Verifier

The canonical verifier, $\mathsf{Ver}_k()$, works by comparing the received tag, t, with a locally calculated tag, t', based on the received message, m.

$$t \stackrel{?}{=} \mathsf{Tag}_k(m)$$

This seemingly benign formula can prove to be fatal for any provably secure message authentication scheme.

3.2.1 Security Analysis

A naive software engineer might look at the canonical verifier and choose to implement it in several obvious ways: a for-loop to compare byte-by-byte, using string.compare(), or simply string1 == string2. However, if any of these methods are used, a difference in run-time of the verifier would be detectable by an adversary. Let s be the time it takes to compare a single byte. If there is a mismatch in the first compared byte, Ver would return in time s. However, if the first compared byte is correct and the second is incorrect, Ver would return in time $2s$ on average.

To turn this timing information into an attack, an adversary could submit an interesting message, m_1, and random tag, $t_1 = \{0xb, ..., 0xb\}^{n/8}$, where $0xb \in \{0,1\}^8$. Then they measure the time it takes for the verifier to return, s. Then, the adversary could increase the first byte of t_1 by 1 ($0x(b+1)$) and submit the new tag $\{0x(b+1), ..., 0xb\}^{n/8}$ with m_1 and measure the time it takes for the verifier to return. If the newly measured time has increased by s, then the adversary can be confident that the first byte of the tag should be $0x(b+1)$, and if the newly measured time has decreased by s, the adversary can be confident that the first byte of the tag should be $0xb$. If the measured time is the same, increase the first byte of T_r by 1 again ($0x(b+2)$) and repeat until the verifier takes about twice as long. In this way, an adversary can search $0 <= i < 256$ through each byte of the tag in order to come up with a forgery. The complexity of such an attack is linear : $O(n/8 * 256) = \text{poly}(n)$ where n is the security parameter (number of bits in the tag).

3.2.2 Side-Channel Attack

Our side-channel exploit was implemented using a client-server socket interface. The server performs $Tag(m)$ and $Ver(t)$ using our own CBC-MAC implementation based on 128-bit AES in CBC-mode, and we used a for-loop to implement the canonical verification function. Additionally, Ver returns when the first difference is found and contains a 10 ms sleep statement that gets executed when a correct comparison is made. The delay is intented to help separate signal from noise, and in practice statistical analysis could be used on the client-side instead. Additionally, an adversary might try to induce interrupts on the server which would have the same affect as our delay.

To forge a tag, our client constructs an arbitrary message, m_1 and an arbitrary tag, t_1. The client sets the first byte of t_1 to 0 then sends $t_1 \| m_1$ to be verified by the server. When the server responds, the client checks the measured wall time to determine whether the current call took more time than a specified threshold. If so, it's likely that was the correct modified byte. To improve robustness, the client double checks the 'candidate' tag-message pair by sending it to the server again to see if the time difference is similar. When the client is confident the time difference is significant, it keeps the 'candidate' byte as the 'forgery' byte and moves to the next byte in t_1 to start the process again until all 16 bytes have been guessed. On average, this attack takes less than 8 minutes to come up with a forgery when using localhost interface. In practice, network latency and jitter would be a hurdle for this attack causing it to be harder to differentiate time differences and make the overall attack take longer. Some of the defenses we thought of were to define a constant-time comparator by always looping through the length of the tag even after a mismatch has been found. Another is to compute the hash of the received tag and calculated tag, then compare if the hashes are equal in constant time.

To run, first start the server:

```
$> ./server
```

Then, in a new terminal start the client using the localhost interface against MAC:

```
$> ./side_channel localhost MAC
```

Source code can be found in *src/side_channel.cpp*.

4 Authenticated Encryption (AE) Schemes

Secrecy and integrity are two properties often required by a security system. A naive software engineer might wrongly assume that since the building blocks for secrecy and integrity are well-established it is just a matter of stacking them together to construct an AE scheme. However, the following sections demonstrate concrete attacks against the two wrong ways to build AE schemes.

4.1 Encrypt-and-Tag (EAT)

An Encrypt-and-Tag scheme takes the following form:

$$\mathsf{Enc}(m) = t \| c \text{ where } t = \mathsf{Tag}_{k_a}(m) \text{ and } c = \mathsf{Enc}_{k_e}(m)$$
$$\mathsf{Dec}(t \| c) = m \text{ where } m = \mathsf{Dec}_{k_e}(c) \text{ and } \mathsf{Ver}(t, \mathsf{Tag}_{k_a}(m))$$

Notice that Alice applies a secure encryption scheme to the message and a secure message authentication scheme to the message, then sends the two pieces, t and c, to Bob. In order to verify the message, Bob must first decrypt it because the tag was calculated on the plaintext. For simplicity, Alice sends $t \| c$, but in practice this is not necessary - they may be sent as part of a more complex protocol or independently; what's important is that the adversary can obtain both.

4.1.1 Security Analysis

We have identified two main reasons EAT is insecure: (1) The ciphertext is not authenticated which opens the scheme up to chosen-ciphertext attacks; and (2) The tag produced directly from the message has no secrecy guarantees which can compromise the secrecy of the message.

To attack the underlying encryption scheme, we have applied our padding oracle attack as in Section 2.1 to successfully uncover the plaintext. Moreover, *any* encryption scheme used in EAT that is not CCA secure will reveal the plaintext because decryption is carried out directly on the input ciphertext that an adversary may control!

To attack the underlying authentication scheme, the two forgery attacks described in Section 3 can be used. The first takes advantage of an implementation flaw specific to CBC-MAC while the second takes advantage of a timing side-channel in the canonical verifier. Both would enable forgeries on arbitrary ciphertexts.

In order for the AE scheme to be completely compromised, a meaningful ciphertext would need to be constructed and a tag forged. While a meaningful ciphertext can't be easily constructed, an active MitM could intercept and prevent several messages intended for Bob. After exploiting a CCA weakness against the scheme, the plaintext could then be recovered and the adversary could forge a fresh tag on whichever message they want to send by exploiting the authentication scheme. This is not unreasonable in a real-world setting because attaining an active MitM is easy on an IP network [7].

4.1.2 Attack Implementation

To carry out the attacks mentioned above, we implemented a client-server interface using sockets in C++. The server implemented 128-bit AES in CBC-mode for the underlying encryption scheme and 128-bit AES in CBC-mode to construct the tag. The details for the client attack against the underlying authentication scheme can be found in Section 3.2.

To run, first start the server:

```
$> ./server
```

Then, in a new terminal start the client using the localhost interface against EAT:

```
$> ./side_channel localhost EAT
```

Additionally, the attack from Section 2.1 will uncover the complete plaintext due to the padding oracle CCA weakness present in the underlying encryption scheme. However, due to time constraints we did not port this attack client to the socket API.

4.2 Tag then Encrypt

Tag then Encrypt (TTE) is the second form of insecure authenticated encryption. It has the following form:

$$\mathsf{Enc}(m) = c \text{ where } c = \mathsf{Enc}_{k_e}(t\|m) \text{ and } t = \mathsf{Tag}_{k_a}(m)$$
$$\mathsf{Dec}(c) = m \text{ where } t\|m = \mathsf{Dec}_{k_e}(c) \text{ and } \mathsf{Ver}(t, \mathsf{Tag}_{k_a}(m))$$

For encryption, the message is first tagged, then the resulting tag, t, is concatenated to the original message, m, and the pair $t\|m$ is encrypted to produce a ciphertext, c. For decryption, the ciphertext is first decrypted, then the tag and message are split into t and m. Finally, $Ver(t, Tag_{k_a}(m))$ is carried out before the message is passed to the application.

4.2.1 Security Analysis

While TTE somehow seems better than EAT in terms of secrecy because a tag is not sent in the clear and therefore cannot give any information about the plaintext, it still suffers from the problem that an adversary can submit any ciphertext, c, that will be unwittingly decrypted by the recipient. So any underlying encryption scheme that is not CCA secure can have its plaintext completely revealed. Also, if the underlying encryption scheme is CTR-mode (for example), an adversary can change bits of the tag by changing corresponding bits in the ciphertext to effectively carry-out a forgery attack on the underlying authentication scheme. Unfortunately due to time constraints we weren't able to fully implement attacks against TTE, but the same attacks against EAT encryption schemes work against TTE because the ciphertext can be manipulated by an adversary enabling CCA attacks.

The best defense is to use Encrypt then Tag, because in that scheme, the ciphertext is authenticated so the decryption doesn't happen unless the ciphertext is authentic.

5 Conclusion

Symmetric key block ciphers are an important and powerful cryptographic tool. They are faster and more efficient than asymmetric schemes, use relatively simple constructions, some can be parallelized, and they are quantum-resistant. However, as we have shown, they are not without fault, and they can be tricky to get right. By developing our own implementations of block-cipher schemes we learned that even with secure libraries such as libcrypt from OpenSSL, it is easy to implement an *insecure* cryptosystem. Several factors contributed to the difficulty in getting it right: OpenSSL documentation is terrible [9][8]; Working with raw character arrays and unsigned chars in C++ on top of sockets is error prone and difficult to debug; and vulnerabilities in cryptosystems are not always obvious and sometimes are not apparent until an attack is implemented against the system.

Attacks that reach outside the bounds of security assumptions provided by the algorithm (i.e. side channel attacks or padding oracle attacks) are difficult to defend against and requires careful forethought that is often not present when the implementations are designed and written. Experience and instruction dictates that the best way to make sure a system is secure before deployment is to try to attack it yourself. Putting oneself in the mindset of the attacker opens a perspective that is not present when simply implementing a cryptosystem.

By implementing our own cryptosystems and attacks, we have gotten a taste of how to think like an attacker during the software development stage, identified security flaws in the most common block-cipher modes, and learned various ways to defend against attacks that exploit these flaws.

References

[1] R. Fedler. *Padding Oracle Attacks* semanticscholar.org, 2013. Accessed 12/8/16

[2] J. Katz and Y. Lindell *Introduction to Modern Cryptography* 2nd ed. CRC Press, 2015.

[3] NIST *FIPS 197 Advanced Encryption Standard (AES)*, 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Accessed 12/1/2016.

[4] Github.com *OpenSSL : TLS/SSL and crypto library*, https://github.com/openssl/openssl. Accessed 12/1/2016.

[5] Github.com *English Words*, https://github.com/dwyl/english-words. Accessed 12/8/2016

[6] OpenSSL.org *OpenSSL*, https://www.openssl.org/. Accessed 12/1/16.

[7] D. Song *dSniff*, https://en.wikipedia.org/wiki/DSniff. Accessed 12/12/16.

[8] Patrick *Just How Bad is OpenSSL?*, https://lists.randombit.net/pipermail/cryptography/2012-October/003388.html. Accessed 12/14/16.

[9] P. Kamp *Please Put OpenSSL out of Its Misery* ACM. 2014.