2011 IEEE Symposium on Security and Privacy

Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET

Thai Duong
Vnsecurity/HVAOnline
Ho Chi Minh City, Vietnam
thaidn@vnsecurity.net

Juliano Rizzo
Netifera
Buenos Aires, Argentina
juliano@netifera.com

***Abstract*—This paper discusses how cryptography is misused in the security design of a large part of the Web. Our focus is on ASP.NET, the web application framework developed by Microsoft that powers 25% of all Internet web sites. We show that attackers can abuse multiple cryptographic design flaws to compromise ASP.NET web applications. We describe practical and highly efficient attacks that allow attackers to steal cryptographic secret keys and forge authentication tokens to access sensitive information. The attacks combine decryption oracles, unauthenticated encryptions, and the reuse of keys for different encryption purposes. Finally, we give some reasons why cryptography is often misused in web technologies, and recommend steps to avoid these mistakes.**

***Keywords*-Cryptography, Application Security, Web security, Decryption oracle attack, Unauthenticated encryption.**

I. Introduction

At EuroCrypt 2004 Nguyen asked, "How can one know if what is implemented [in software] is good cryptography?" [1]. This is an important question because history has shown that cryptography is often used incorrectly in both open source and proprietary software (see [1]–[7]). Nevertheless, despite the important role of the WWW, there is limited research available from both the cryptographic and web security communities to answer Nguyen's question for the case of cryptographic implementations in web technologies.

This paper shows that badly implemented cryptography is not limited to traditional software, but is highly pervasive in web applications as well. Since HTTP is a stateless protocol, web developers must either manage the user session state data on the server or push it to the client. For performance and scalability reasons, web developers tend to go with the latter method. They want to keep session information secret, so they correctly turn to cryptography. However, implementing crypto is error-prone. We observe that unauthenticated encryption is often used to encrypt session state data such as HTTP cookies and view states. Unauthenticated encryption is dangerous [7]–[11], particularly when used in an authentication system. The ability to forge a ciphertext that decrypts to a desired plaintext allows the attacker to impersonate other users easily [7]. Web developers also tend to use the same keys for different encryption purposes. These cryptographic errors together make the Web become a gold mine for chosen-ciphertext attacks.

In this paper, we illustrate this point by examining the case of cryptographic implementations in web applications based on ASP.NET [12]. The framework was first released in January 2002 with version 1.0 of the .NET Framework. As of September 2010, it is believed that 25% of all the Internet web sites are developed using ASP.NET.[1]Here we review ASP.NET v4.0, which was the current stable version at the time of submission. Our comments also apply to several previous versions of ASP.NET.

We observe several cryptographic flaws in ASP.NET v4.0. The most serious flaw (which turns out to have been present in ASP.NET for almost three years) is a consequence of unauthenticated encryption. We present two practical and highly efficient attacks that allow attackers to steal cryptographic secret keys, forge authentication tokens and destroy the security model of every ASP.NET v4.0 application. Both are chosen-ciphertext attacks that combine decryption oracles similar to the padding oracle introduced by Vaudenay at EuroCrypt '02 [13] and the CBC-R technique that Rizzo and Duong demonstrated at USENIX WOOT '10 [14]. The novelty of these attacks is that not only can the attacker decrypt secret data in ASP.NET, but he also can create ciphertexts that after being decrypted and processed by ASP.NET, allow him to retrieve sensitive information.

The rest of the paper is organized as follows. In Section II, we give an overview of ASP.NET v4.0 and the cryptographic vulnerabilities in the framework. In Section III, we provide sufficient background on decryption oracle attacks and the CBC-R technique to make the paper self-contained. In Section IV, we describe our first attack exploiting padding oracles in the framework. In Section V, we describe our second attack, which is faster than the first attack and does not require a padding oracle. In Section VI, we consider the practical impact of our attacks as well as countermeasures that prevent them. Our reflections on why cryptography is often misused in web technologies and our recommendations can be found in Section VII.

[1]See http://trends.builtwith.com/framework.

1081-6011/11 $26.00 © 2011 IEEE
DOI 10.1109/SP.2011.42

II. An Overview of ASP.NET

In this section, we review some key concepts and terminology for ASP.NET. We then describe how the framework misuses cryptography when attempting to tamper-proof and encrypt sensitive information.

A. Key Concepts and Terminology

Machine Key: The machine key is a pair of global secret keys set in the web application configuration to be used for encryption and authentication. A key named `validationKey` is used to generate hashed message authentication codes (HMAC) to protect the integrity of authentication tickets and view states. A second key named `decryptionKey` is used to encrypt and decrypt authentication tickets and view states.

View State: An ASP.NET application is a collection of .NET pages, known officially as "web forms". ASP.NET applications are hosted by a web server and are accessed using the stateless HTTP protocol. As such, if an application uses stateful interaction, it has to implement state management on its own. ASP.NET provides various functions for state management, and view state is one of them.

View state refers to the page-level state management mechanism utilized by the HTML pages emitted by ASP.NET applications to maintain the state of the web form controls and widgets. The state of the controls is sent to the server at every form submission in a hidden field known as `__VIEWSTATE`. The main use for this is to preserve form information when the page is reloaded. The hidden field is updated by the server and is never modified by the client.

By default, the `validationKey` is used to generate an HMAC from the view state content. This HMAC is stored as a hidden field in ASP.NET forms, and is verified on every request. If ASP.NET receives a request with an invalid HMAC, the request is dropped. Because the view state can contain sensitive data, ASP.NET allows developers to enable view state encryption on a server-wide or per-page basis. Microsoft's documentation on view state encryption is unclear as to whether the view state is still authenticated if encryption is enabled.[2] Based on our testing, we see that ASP.NET v4.0 either authenticates or encrypts view states, but it does not apply both operations at the same time.

Forms Authentication Tickets: Since ASP.NET aims to become a rapid web development framework, it provides built-in solutions for many common problems in web development. One of them is user account support. Providing user account support for any site involves the same set of steps: creating a datastore, a login page and a register page; defining authentication and authorization mechanisms; creating a page for the site's administrators to manage the user accounts; and so forth. Prior to ASP.NET, developers had to decide how to implement all of these features on their own. To ease this burden, ASP.NET introduced the concept of forms-based authentication. This feature provides a `FormsAuthentication` class that handles signing in and out of a site, as well as a protected authentication ticket to remember users' login states across page requests.

Forms authentication uses an authentication ticket that is created when a user logs on to a site; this ticket is then used to track the user throughout the site. The forms authentication ticket is created by the `FormsAuthentication` class as follows. Once the user is validated, the FormsAuthentication class internally creates a `FormsAuthenticationTicket` object by specifying his username; the version of the ticket; the directory path; the issue date of the ticket; the expiration date of the ticket; whether the ticket should be persisted; and, optionally, user-defined data. Next the `FormsAuthenticationTicket` object is serialized, then an HMAC is generated from the serialized data using the `validationKey`. This HMAC is appended to the end of the serialized data, then the whole content is encrypted using AES or DES with the `decryptionKey`. The resulting string is called the form authentication ticket, and it is usually contained inside an HTTP cookie. However, ASP.NET supports cookie-less forms authentication; in this case the ticket is passed in a query string.

Each time a subsequent request is received after authentication, the `FormsAuthenticationModule` class retrieves the authentication ticket from the authentication cookie or the query string, decrypts it, computes the hash value, and verifies the HMAC value to ensure that the ticket has not been tampered with. Finally, the expiration time contained inside of the forms authentication ticket is verified. If all checks pass, ASP.NET will authenticate the request, and the user is authenticated as the username contained in the authentication ticket. Consequently, the ability to create valid authentication tickets is sufficient for an attacker to impersonate any user account in ASP.NET applications.

Web Resources and Script Resources: In the .NET framework, an assembly is a compiled code library used for deployment, versioning and security. An assembly consists of one ore more files. These files can be code modules, web resources (e.g., HTML, CSS, or images), or script resources (e.g., Javascript). Web developers reference these static resources through a standard API.

Web and script resources rely on special handlers named `WebResource.axd` and `ScriptResource.axd`, respectively, to serve resources to the web browser. When a request comes in from the client for `WebResource.axd`, the handler looks for the web resource identifier in the

[2] See http://msdn.microsoft.com/en-us/library/ff649308.aspx.

`QueryString` method of the `Request` object. Based on the value of the web resource identifier, the handler then tries to load the assembly that contains this resource. If this operation is successful, the handler will then look for the assembly attribute and load the resource stream from the assembly. Finally, the handler will obtain the data from the resource stream and send it to the client together with the content type specified in the assembly attribute.

The request format for both `WebResource.axd` and `ScriptResource.axd` is as follows:

WebResource.axd?d=*encrypted_id*&t=*timestamp*

We observe two interesting things about the `d` parameter:

1) ASP.NET encrypts this parameter, but does not authenticate the ciphertext.
2) Due to a feature in `ScriptResource.axd`, an attacker can download arbitrary files inside the document root of ASP.NET applications given a valid encrypted `d` parameter.

B. Cryptographic Design Flaws in ASP.NET

We observe two sets of cryptographic flaws in ASP.NET: improper use of cryptographic primitives, and insecure key management.

Insecure Key Management: There are three issues in how ASP.NET manages cryptographic keys.

The first issue is the reuse of keys for different purposes. In the last section, we showed that the framework uses cryptography to authenticate and encrypt view states, forms authentication tickets, web resources and script resources. These are pieces of information with different levels of importance. Forms authentication tickets and view states are critical to the security of ASP.NET, but web resources and script resources identifiers do not include very sensitive information. ASP.NET, however, encrypts all of them with the same cryptographic keys.

The second issue is insecure key storage. By default, plaintext cryptographic keys are stored in a file named `web.config` in the document root of ASP.NET applications. In other words, all it takes to steal these keys in any ASP.NET application is one file disclosure.

The last issue is that key management is left to developers and users. Since ASP.NET provides no easy way to generate or revoke keys, users tend not to change keys during the lifetime of an application. Furthermore, it is sometimes impossible to change keys because they are used to encrypt important information that is needed by the applications to operate properly. Users also typically don't change default keys in applications downloaded from the Internet or installed by a third party. When forced to generate keys, it is not uncommon to see users generating their keys using online tools. Websites to generate cryptographic keys are popular amongst ASP.NET developers and users.[3]

Improper Use of Cryptographic Primitives: There are two issues in the way ASP.NET uses cryptography.

First, the cryptographic API in ASP.NET does not use authenticated encryption by default. In Section II-A, we showed that web resources and script resources identifiers are encrypted without authentication.

Secondly, the framework uses the MAC-then-Encrypt mode for authenticated encryption. As previous work has demonstrated, this mode is vulnerable to chosen-ciphertext attacks [9], [15], [16].

III. Decryption Oracle Attacks

In this section, we discuss decryption oracle attacks and the CBC-R technique. In this and subsequent sections, we follow the notation described in Section 4 of [17]. It is important to stress that the padding oracle is just one kind of decryption oracle, and we have found decryption oracles that are easier and faster to exploit in ASP.NET. We illustrate this point in Section V.

A. The Padding Oracle Attack

The padding oracle attack was first introduced by Vaudenay at EuroCrypt '02 [13]. As explained in Paterson and Yau's summary [18], the padding oracle attack requires an oracle that, on receipt of a ciphertext, decrypts it and replies to the sender whether the padding is valid or invalid. The attack works under the assumption that the attackers can intercept padded messages encrypted in CBC mode and have access to the aforementioned padding oracle. The result is that attackers can recover the plaintext corresponding to any block of ciphertext using an average of $128 \star b$ oracle calls, where b is the number of bytes in a block.

1) Padding Oracles In ASP.NET: There are several padding oracles in default components of the framework. They are all application independent, (i.e. they exist in every ASP.NET application). We divide them into two different sets:

1) Authenticated encryption padding oracles: as discussed in Section II, ASP.NET uses the MAC-then-Encrypt mode to protect form authentication tickets. Since this mode is vulnerable to chosen-ciphertext attacks, we have a padding oracle here. Beside forms authentication tickets, ASP.NET also uses MAC-then-Encrypt for role cookies and anonymous identification that can also be used as padding oracles.[4]

[3] See http://aspnetresources.com/tools/machineKey

[4] See http://msdn.microsoft.com/en-us/library/ff649308.aspx.

2) Unauthenticated encryption padding oracles: as noted in Section II, ASP.NET encrypts the references to script and web resources, but it does not protect the produced ciphertext with an authentication code. This introduces additional padding oracles into the framework. We will use them in our attacks described in Section IV and Section V.

Although we are going to describe more powerful attacks in this paper, attackers can use these padding oracles to decrypt and obtain secrets from view states, form authentication tickets, and other encrypted information in ASP.NET applications.

B. Turning Decryption Oracles into Encryption Oracles

In this section, we review CBC-R, a technique to turn a decryption oracle into an encryption oracle. First introduced by Rizzo and Duong [14], this technique is important because it allows attackers to create valid ciphertexts that are trusted by the target. When a system assumes that a meaningful meassage obtained from the decryption of some ciphertext implies a trusted origin of it, the CBC-R technique allows attackers to create arbitrary ciphertexts to abuse the system.

1) CBC-R: The CBC mode is defined as follows:

CBC Encryption:

$$C_1 = CIPH_K(P_1 \oplus IV);$$
$$C_i = CIPH_K(P_i \oplus C_{i-1}) \qquad \text{for i=2,...,n.}$$

CBC Decryption:

$$P_1 = CIPH_K^{-1}(C_1) \oplus IV;$$
$$P_i = CIPH_K^{-1}(C_i) \oplus C_{i-1} \qquad \text{for i=2,...,n.}$$

CBC-R turns a CBC decryption oracle into a CBC encryption oracle. The process is simple. First, the attacker chooses a random ciphertext block C_i. He then sends C_i to the decryption oracle $\mathcal{O}$ to get its intermediate plaintext. Since

$$P_i = \mathcal{O}(C_i) \oplus C_{i-1}$$

and the attacker can change C_{i-1}, he can make P_i equal to any arbitrary value. Suppose he wants to make P_i equal to some P_x. Then, all he needs to do is to set

$$C_{i-1} = P_x \oplus \mathcal{O}(C_i).$$

But does this make C_{i-1} decrypt to a garbled block P_{i-1}? Yes, but the attacker can fix P_{i-1} by sending C_{i-1} to the decryption oracle to get its intermediate plaintext, and set

$$C_{i-2} = P_{i-1} \oplus \mathcal{O}(C_{i-1}).$$

Now, the attacker has two consecutive plaintext blocks P_{i-1} and P_i of his choice, and a leading garbled block P_{i-2} that he can correct by inserting a new ciphertext block C_{i-3}. By repeating this operation, he can efficiently encrypt a complete message block by block, starting from the last one. Since the first block of the CBC ciphertext stream depends on the IV, if the attacker can set the IV, then the decrypted data will be exactly what he wants without any garbled blocks. If the attacker doesn't control the IV, then the first block is garbled. For an overview of this process, refer to Algorithm 1.

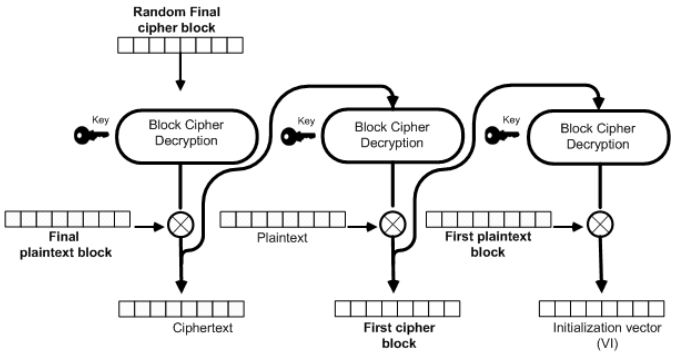

Figure 1. CBC-R.

Algorithm 1 CBC-R.

1) Choose a plaintext message P, pad the message, and divide it into n blocks of b bytes denoted by $P_1, P_2, \ldots, P_n$.
2) Pick a few random bytes $r_1, r_2, \ldots, r_b$, and set $C_n = r_1|r_2|\ldots|r_b$.
3) For $i = n$ down to 1:
   $C_{i-1} = P_i \oplus \mathcal{O}(C_i)$
4) Set $IV = C_0$.
5) Output IV and $C = C_1|\ldots|C_n$.

2) CBC-R Without Controlling IV: We have shown that CBC-R allows the attacker to encrypt any message. But, if he cannot set the IV, then the first plaintext block will be random and meaningless. If the victim expects the decrypted message to start with a standard header, and the attacker doesn't control the IV, then the victim will ignore the forged message constructed by CBC-R. This is what happens with the resource identifiers in ASP.NET, where the first two characters of the decrypted identifiers must be in the limited set of defined options. We have found two workarounds.

Using Captured Ciphertexts as Prefix: If the attacker captures a ciphertext whose plaintext is a valid message, then he can prepend the ciphertext to his CBC-R encrypted message

Algorithm 2 Brute-forcing C_1.

1) Choose a plaintext message P, pad the message, and divide it into n blocks of b bytes, denote them $P_1, P_2, \ldots, P_n$ where $P_1 = P_{header}|P_1^*$.
2) Use CBC-R and the oracle to build $C_n, \ldots, C_2, C_1$ so that $C_1|C_2|\ldots|C_n$ decrypts to $P_{garbage}|P_2|P_3|\ldots|P_n$.
3) Pick a few random bytes $r_1, r_2, \ldots, r_b$, and set $C_1 = r_1|r_2|\ldots|r_b$.
4) Test if $C_1|C_2|\ldots|C_n$ decrypts to $P_{header}|P_{garbage}|P_3|P_4|\ldots|P_n$. If not, go back to step 3.
5) Output $C = C_1|C_2|\ldots|C_n$.

to get a valid header after decrypting:

$$P_{valid} = CIPH_K^{-1}(C_{known}|IV_{CBC-R}|C_{CBC-R}).$$

While the resulting forged plaintext message will have a valid header, it still has a garbled block at the position of IV_{CBC-R}. This broken block causes the victim to reject the message, unless the attacker carefully chooses a prefix such that the garbled block becomes part of some component that doesn't affect the semantics of the message.

Brute-Forcing C_1: The attacker can also brute-force the first block C_1. In CBC-R, the final block C_n is a random block. Each different C_n yields a different $C_{n-1}, ..., C_1$ chain. In other words, CBC-R can produce many different ciphertexts that decrypt to the same plaintext block chain $P_n, ..., P_2$. The only difference is the first plaintext block, which is computed as follows:

$$P_1 = CIPH_K^{-1}(C_1) \oplus IV.$$

The attacker wants P_1 to contain a valid header. In some systems, this means that the first few bytes of P_1 must match some magic numbers. There are also systems that accept a message if the first byte of P_1 matches its size. If this is the case, the attacker can try his luck by brute-forcing C_1. One way to do this is to change C_n, hence changing C_1, until a valid P_1 is found. A faster technique is to use CBC-R to build a chain $C_{n-1}, ..., C_2$, then generate a random C_1 until a valid P_1 is found. For example, if the first byte of P_1 must match the message size, trying an average 256 different C_1 is enough to obtain a valid message. Or, if the first two bytes of P_1 must match some magic numbers, then we need on average 2^{16} different C_1 to get a valid message. For an overview of this process, refer to Algorithm 2.

C. Using CBC-R to Attack ASP.NET

As noted in the description of the `ScriptResource.axd` handler in Section II, if an attacker supplies a valid encrypted `d` parameter to the handler, he can abuse the file retrieval functionality to download any file located inside the root directory of ASP.NET applications. Downloading the `web.config` file is the best first step for an attacker. This file is present in most applications and contains important secrets, including the keys necessary to forge authentication tickets and database passwords.

In order to download files, the attacker has to craft a `d` parameter that decrypts to a string with the following format

```
R#anything|||~/path/to/file
```

The first two bytes can be one of these four values `r#`, `R#`, `q#`, and `Q#`. This is a perfect application for CBC-R. The attacker can use the method of Section III-B2 and Algorithm 2 to construct `d` with a three block message so that the last two blocks will be decrypted to `garbage|||~/path/to/file`. He then brute-forces the first block until he gets one of the magic byte values.

IV. First Attack: CBC-R with Padding Oracles

As discussed in Section III-A1, there are many application independent padding oracles in ASP.NET. Each of them can be used together with CBC-R to construct the `d` parameter as described in Section III-C. In this first attack, we will use the padding oracle in `WebResource.axd`.

A. Attack Implementation

First, the attacker has to figure out a reliable way to detect padding information from the `WebResource.axd` responses. He can easily do this because `WebResource.axd` will return a 500 HTTP response if the padding is invalid and a 404 HTTP response otherwise.

Second, the attacker needs to build a block decryption algorithm similar to the one described by Vaudenay [13]. The attacker constructs a three-block message $= C_{valid}|C_{random}|C_{target}$. C_{target} is the block that he wants to decrypt. C_{valid} is a valid resource identifier that is easily found in ASP.NET applications. The attacker needs C_{valid} to ensure that the decryption of his message is correctly formatted as a resource identifier, or ASP.NET would return a 500 HTTP response even if the padding is valid. The attacker changes C_{random} until he gets a 404 HTTP response, which indicates that a valid padding has been found. The attacker then uses Vaudenay's last-byte-decryption algorithm to decrypt the last byte of C_{target}. After obtaining the last byte, the attacker changes C_{random} again, and uses Vaudenay's block decryption algorithm to obtain the rest of C_{target}.

Third, after the attacker has a reliable implementation of the block decryption algorithm, he can use CBC-R with the approach used in Algorithm 2 and Section III-C to build the `d` parameter. See Algorithm 3 for a pseudo-implementation of this attack.

Algorithm 3 Downloading files using CBC-R and the padding oracle in `WebResource.axd`.

1) Pad the messsage `|||~/path/to/file` according to the PKCS#5 padding scheme, and set the whole result as the target P.
2) Divide P into n blocks of b bytes, denote them $P_1, P_2, \ldots, P_n$.
3) Use CBC-R to build $C_n, \ldots, C_2, C_1$ so that $C_1|C_2|\ldots|C_n$ decrypts to $P_{garbage}|P_1|P_2|\ldots|P_n$.
4) Pick a few random bytes $r_1, r_2, \ldots, r_b$, and set $C_0 = r_1|r_2|\ldots|r_b$.
5) Set $d = C_0|C_1|...|C_n$, and send it to `ScriptResource.axd` to see if `/path/to/file` is downloaded. If no, go back to step 4.
6) Output d.

B. Attack Cost

As described in Algorithm 3, there are two independent steps in a CBC-R attack against ASP.NET. The first step is to construct $C_1|C_2|...|C_n$, and the second step is to brute-force C_0. Although here we only perform a cost analysis for the case where the attacker wants to download `web.config`, the same method can be applied to other files.

Suppose the block size b is 16. The attacker needs just one block for `|||~/web.config` (which is 15 bytes). For each block, the attacker needs on average $128 * b$ oracle calls for the first step. For the second step, since the first two bytes can be one of the four values `r#`, `R#`, `q#`, and `Q#`, the attacker needs on average 2^{14} HTTP requests to brute-force the first two bytes of C_0. Since each oracle call is equal to an HTTP request sent to the web server running ASP.NET, the attack needs on average $2^{14} + 128 * b = 2^{14} + 128 * 16 = 18432$ HTTP requests. In the next section, we describe a faster attack that does not need a padding oracle.

V. Faster Attack: CBC-R with T-block Decryption Oracle

A. The T-block Decryption Oracle in ScriptResource.axd

Besides the padding oracle in `WebResource.axd`, we found another application independent decryption oracle in `ScriptResource.axd`. If the first byte after decrypting the `d` parameter is "T", the handler will send all of the decrypted data to the client. In order to trigger this decryption oracle, the attacker has to find a T-block whose first byte after decryption is "T". The T-block decryption oracle is much faster than any padding oracle because it requires only one HTTP request to decrypt several blocks.

There is one minor issue with this approach. Before sending the decrypted data to the client, `ScriptResource.axd` performs a character encoding conversion. The conversion function is non-injective, so that the attacker cannot recover all the bits of the output of the T-block oracle. Although we tried reversing the framework to understand the conversion function, it was not possible to avoid losing a significant number of bytes. In this sense, we consider the T-block oracle to be a kind of "noisy oracle" with a high signal-to-noise ratio. In the next section, we describe a solution to this problem.

Algorithm 4 Find a T-block.

1) Find a known ciphertext in the target application, denote its last two blocks as C^1_{known} and C^2_{known}.
2) Pick a few random bytes $r_1, r_2, \ldots, r_b$, and set $T = r_1|r_2|\ldots|r_b$.
3) Set $d = T|C^1_{known}|C^2_{known}$, and send it to `ScriptResource.axd` to see if the HTTP response code is 200. If not, go back to step 3.
4) Output T.

B. Attack Implementation

The attacker first needs to find a T-block whose first byte after decryption is "T". The attacker generates a random block and sends that block to `ScriptResource.axd` to see if he gets an HTTP 200 response with HTML content including the decrypted data. Since `ScriptResource.axd` decrypts its input, the attacker needs to ensure that the output is properly padded (or else he never gets a HTTP 200 response). The attacker can ensure a valid padding by appending the last two blocks of any known ciphertext to what he sends to `ScriptResource.axd`. Algorithm 4 describes this process.

After getting a T-block oracle, the next steps are to build the CBC-R $C_n, C_{n-1}, \ldots, C_1$ chain and to find C_0 as described in Section III-C. Since we lose some bytes in the output of the T-block oracle, we need to figure out how to recover them. What we describe in the next paragraph may not be the best possible solution to this issue, but it is efficient enough for the purpose of using the T-block as part of CBC-R.

As described in Section V-A, some bits of the plaintext are lost in the T-block oracle responses. The attacker needs to decrypt complete blocks to be able to encrypt using CBC-R. An efficient decryption process also accelerates the exhaustive search of the prefix required to download files.

The character conversion function causes two problems:

- Plaintext bytes with values larger than 0x7F are converted to the Unicode replacement character which in UTF-8 is encoded as the three bytes 0xEF 0xBF 0xBD.
- Some pairs of bytes are converted to a single Unicode replacement character.

Algorithm 5 Downloading files using CBC-R and the T-block Decryption Oracle in `ScriptResource.axd`.

1) Pad the messsage `|||~/path/to/file` according to the PKCS#5 padding scheme, and set the whole result as the target P.
2) Divide P into n blocks of b bytes, denote them $P_1, P_2, \ldots, P_n$.
3) Use Algorithm 4 to find a T-block, denote it as T.
4) Use T to find a C_0 whose the first two bytes after decryption match one of the magic byte values.
5) Use CBC-R with T to build $C_n, \ldots, C_2, C_1$ so that $C_1|C_2|\ldots|C_n$ decrypts to $P_{garbage}|P_1|P_2|\ldots|P_n$.
6) output $d = C_0|C_1|\ldots|C_n$.

The attacker can solve the first problem by modifying a ciphertext block to alter selected bits in the decryption of the next block. To decrypt a block C_i, the attacker can send a four-block message in a format of $C_r|C_i|\tilde{C_r}|C_i$ where C_r is a random block and $\tilde{C_r}$ is the result of changing the highest bit of all bytes in C_r (i.e., he can XOR each byte with 0x80). The decryption process of the CBC mode ensures that if a byte in the decryption of the first C_i is larger than 0x7f, then the byte at the same position in the decryption of the second C_i will be smaller than or equal to 0x7f. The attacker can then get the full plaintext of C_i by combining the bytes that he gets from the decryptions of the two C_i.

One way to solve the second problem would be to repeatedly perform requests to the oracle, changing part of the input, until the output has as many bytes as expected. A more efficient aproach is to construct a large query such as $C_1|C_{known}|C_2|C_{known}|C_3|C_{known}|\cdots|C_n|C_{known}$, where C_i are groups of blocks that the attacker wants to decrypt and $C_{known} = C^1_{known}|C^2_{known}$, where $C^1_{known}|C^2_{known}$ are the last two blocks of any known ciphertext. Notice that C^2_{known} would be decrypted as a known plaintext P_{known}. The attacker can then split the output of the T-block oracle into chunks using P_{known} as the separator. He then calculates the length of each chunk, ignores chunks that are losing some bytes, and accepts only those chunks that don't lose any bytes. Further optimizations are possible using redudant information present in the incomplete chunks.

With these optimizations, the attacker can use the T-block as an efficient decryption oracle to attack the framework. Algorithm 5 describes how the attacker can use the T-block oracle together with CBC-R to download files.

C. Attack Cost

The cost of attack consists of three parts, which are described in this section. The first part is the cost of finding a T-block, which on average takes 256 HTTP requests.

The second part is the cost of CBC-R. Although here we only perform cost analysis for the case where the attacker wants to download `web.config`, a similar analysis applies to other files. Suppose the block size b is 16. The attacker needs just one block for `|||~/web.config` (which is 15 bytes.) For each block, the attacker needs only one T-block oracle call, which is equal to one HTTP request. So, the cost of this part is negligible.

The third part is to find a block C_0 whose first two bytes after decryption match one of the magic byte values. Since the first two bytes of C_0 can be one of the four values `r#`, `R#`, `q#`, and `Q#`, the attacker needs on average 2^{14} HTTP requests to brute-force the first two bytes of C_0. To reduce the number of requests, the attacker can build a large query that contains the maximum number of blocks that fit in a single HTTP request. The attacker can construct a ciphertext that contains a T-block, then 20 repetitions of $C_r|C_i|\tilde{C_r}|C_i$, separated by 19 pairs of $C^1_{known}|C^2_{known}$, then ending with another pair of $C^1_{known}|C^2_{known}$. Each repetition consumes 6 AES blocks, or 6*16 = 96 bytes. With 20 repetitions, the full ciphertext after conversion to BASE64 [19] is approximately 2048 bytes long, which is the maximum query size allowed by ASP.NET. About half of these 20 repetitions would lose some bytes, so the attacker can decrypt 10 blocks per request. In summary, the attack takes on average about $256+2^{14}/10 = 1894$ HTTP requests, each about 2048 bytes long. In other words, the attacker has to send a total of about 3MB of data to the server on average. Note that for a particular target, the attacker can reuse T-block and C_0 in subsequent attacks to download other files.

VI. Impact and Countermeasures

A. Impact

We have presented two attacks on ASP.NET v4.0. The attacks are highly efficient and have been demonstrated to work under realistic scenarios. The best attack requires less than 2,000 HTTP requests to steal secret cryptographic keys in ASP.NET applications. After obtaining those keys, the attacker can easily create authentication tickets that he can use to sign in to any user account. Most applications have administration interfaces, and if the attacker can gain access to those areas, he can then sign in as an administrator and seize control of the whole application. Aministration interfaces often allow unlimited file uploading, which would allow the attacker to achieve remote code execution.

There is other important information inside `web.config` besides cryptographic keys. Since `web.config` is supposed to be private, users and developers tend to put a lot of sensitive information there. It is not uncommon to see connection strings with usernames and passwords to database stores, mail servers and directory services. We have

even seen `web.config` files that contain the password of a Domain Administrator account in Microsoft Active Directory.

Besides `web.config`, the described attacks can also be used to download source code and other intellectual property. The `App_Code` folder in ASP.NET applications contains source code for shared classes and business objects (for example, .cs and .vb files). Files within `App_Code` and any other files inside the document root of applications can be stolen.

To emphasize the real-world impact of our attacks, we note that ASP.NET is a hugely popular web framework, and it is believed that 25% of the applications online are built using it. However, that number is far higher in the corporate and financial services sectors, and applications such as online banking and e-commerce are prime targets for this attack. DotNetNuke CMS[5], the most popular public ASP.NET application with over 600,000 installations, has been demonstrated to be vulnerable to these attacks. Microsoft also acknowledged that Microsoft SharePoint, one of the most popular enterprise collaboration applications, was also affected. [6] Even microsoft.com was vulnerable.[7] It is also worth noting that Mono, the open source implementation of ASP.NET, was vulnerable to these attacks.[8]

B. Countermeasures

Our attacks, especially the T-block attack, are difficult to detect. After all, all of those thousands of requests sent to the target are encrypted, and none of them cause any suspicious logging errors. In other words, countermeasures such as web application firewalls or network security monitoring can hardly detect, let alone prevent, our attacks.

After initial details of our attacks were released to the public [20], Microsoft released an immediate workaround.[9] The workaround focused on padding oracles, and, therefore, it mitigated part of the first attack. However, it completely failed to address the second attack. After we sent more information to Microsoft, they eventually released a patch that did prevent all of our attacks [21]. It is strongly recommended that ASP.NET users and developers immediately install this official security update to protect against these attacks.

C. Related Work

Black and Urtubia [10] have generalized Vaudenay's attack to other padding schemes and modes of operations, and presented a padding method that prevents the attack. Canvel et al. [22] demonstrated the practicality of padding oracle attacks by implementing an attack against the IMAP protocol when used over SSL/TLS. In a typical setting, the attack recovers the IMAP password within one hour. For many years, this was the most pratical application of the attack published.

Klima and Rosa [23] applied the idea of a "format correctness oracle" (of which the padding oracle is a special case) to construct a PKCS#7 validity oracle. Using this oracle, they were able to decrypt one PKCS#7 formatted ciphertext byte using an average of 128 oracle calls. For the Web, Rizzo and Duong [14] used padding oracle attacks to crack CAPTCHAs as well as to decrypt view states and session state data in various popular web development frameworks.

The most vulnerable software studied in [14] is the JavaServer Faces (JSF) framework. Using CBC-R, an attacker can create malicious JSF view states to attack web applications. However, the security consequences of being able to forge JSF view states depend on how the application uses the view state information.

This paper is the first to describe step-by-step how to use decryption oracles and CBC-R to compromise any application using the ASP.NET framework. The new attacks described are more dangerous, generic and efficient than any previous published results involving padding oracles.

VII. Conclusions

In this paper, we analyze and efficiently exploit several cryptographic flaws in ASP.NET, the widely-used web application framework developed by Microsoft. The most serious vulnerability we discovered is the use of unauthenticated encryption. This vulnerability is exacerbated by the reuse of keys to encrypt data with very different levels of importance. We present two practical and highly efficient attacks that allow an attacker to steal cryptographic secret keys, and impersonate any user account in ASP.NET applications. These attacks are performed by abusing components present in every application developed using the framework. The applications are even more exposed if they use the security features provided by ASP.NET, especially form based authentication.

Cryptography is difficult to implement correctly, and cryptographers often advise non-cryptographers not to develop their own cryptography. But if one looks more closely at the current situation, it is evident that web developers and users do not have much choice. ASP.NET developers still have to figure out on their own how to use cryptographic primitives correctly any time they want to build a secure cryptographic protocol. This is not a problem specific to ASP.NET. Most other popular web development frameworks

[5] See http://www.dotnetnuke.com.
[6] See http://sharepoint.microsoft.com/blog/
[7] See http://forums.asp.net/p/1605099/4096837.aspx.
[8] See http://www.mono-project.com/Vulnerabilities#ASP.NET_Padding_Oracle
[9] See http://weblogs.asp.net/scottgu/archive/2010/09/18/important-asp-net-security-vulnerability.aspx.

do not provide their users and developers easy and secure ways to use cryptography [14]. Popular scripting languages, including Ruby, Python, PHP, provide cryptography libraries as bindings to OpenSSL [24]. While OpenSSL is powerful, it is a low-level library that again requires its users to know how to use cryptographic primitives securely.

Unauthenticated encryption should be considered harmful. This is not just a theoretical problem; rather, unauthenticated encryption has repeatedly led to devastating attacks against real systems. Any cryptographic API should use authenticated encryption whenever its users want to encrypt data. The development and popularization of high-level cryptographic toolkits that include authenticated encryption such as Keyczar [25], Cryptlib [26], and NaCl [27] is the first step to providing secure cryptographic software libraries to the general public. The next step might be the integration of these cryptographic toolkits into mainstream web development frameworks.

Acknowledgments

We are grateful to many people for their help in writing this paper. First of all, we would like to thank William Robertson, Ned Bass and the anonymous reviewers for their work and valuable comments that have significantly improved the quality of our initial manuscript. Our thanks to Michal Trojnara, Giang Duc Pho, Luciano Notarfrancesco, Peter Gutmann, Agustin Azubel, Matias Soler, Agustin Gianni and especially Huong Lan Nguyen for the care with which they reviewed the original draft; and for conversations that clarified our thinking on this and other matters. We would also like to thank Kenneth Paterson and Serge Vaudenay for their encouragement and instruction.

REFERENCES

[1] P. Nguyen, "Can We Trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1. 2.3," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004, pp. 555–570.

[2] I. Goldberg and D. Wagner, "Randomness and the Netscape Browser," *Dr Dobb's Journal-Software Tools for the Professional Programmer*, vol. 21, no. 1, pp. 66–71, 1996.

[3] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," in *Proc. USENIX Security Symp*, 2002, pp. 315–325.

[4] K. Jallad, J. Katz, and B. Schneier, "Implementation of Chosen-Ciphertext Attacks against PGP and GnuPG," *Information Security*, pp. 90–101, 2002.

[5] J. Katz and B. Schneier, "A Chosen-Ciphertext Attack against Several E-mail Encryption Protocols," in *Proceedings of the 9th conference on USENIX Security Symposium-Volume 9*. USENIX Association, 2000, p. 18.

[6] B. Schneier, "Security in the Real World: How To Evaluate Security Technology," *Computer Security Journal*, vol. 15, no. 4, p. 1, 1999.

[7] T. Yu, S. Hartman, and K. Raeburn, "The Perils of Unauthenticated Encryption: Kerberos Version 4," in *Proc. NDSS*, vol. 4. Citeseer, 2004.

[8] M. Bellare, T. Kohno, and C. Namprempre, "Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 2, p. 241, 2004.

[9] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm," *Journal of Cryptology*, vol. 21, no. 4, pp. 469–491, 2008.

[10] J. Black and H. Urtubia, "Side-channel Attacks On Symmetric Encryption Schemes: The Case for Authenticated Encryption."

[11] K. Paterson and A. Yau, "Cryptography in Theory and Practice: The Case of Encryption in IPsec," *Advances in Cryptology-EUROCRYPT 2006*, pp. 12–29, 2006.

[12] ASP.NET, "The Official Microsoft Web Development Framework. http://www.asp.net."

[13] S. Vaudenay, "Security Flaws Induced by CBC Padding-Applications to SSL," in *Advances in Cryptology-EUROCRYPT 2002*. Springer, 2002, pp. 534–545.

[14] J. Rizzo and T. Duong, "Practical Padding Oracle Attacks," *USENIX WOOT*, 2010.

[15] J. An, Y. Dodis, and T. Rabin, "On the Security of Joint Signature and Encryption," in *Advances in Cryptology-EUROCRYPT 2002*. Springer, 2002, pp. 83–107.

[16] H. Krawczyk, "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)," in *Advances in Cryptology—CRYPTO 2001*. Springer, 2001, pp. 310–331.

[17] M. Dworkin, "NIST Recommendation for Block Cipher Modes of Operation, Methods and Techniques," *NIST Special Publication*.

[18] K. Paterson and A. Yau, "Padding Oracle Attacks On the ISO CBC Mode Encryption Standard," *Topics in Cryptology–CT-RSA 2004*, pp. 1995–1995, 2004.

[19] S. Josefsson, "RFC 3548-The Base16, Base32, and Base64 Data Encodings. IETF," 2003.

[20] T. Duong and J. Rizzo, "Padding Oracles Everywhere," *EKOPARTY*, 2010.

[21] Microsoft. Microsoft Security Bulletin MS10-070 - Vulnerability in ASP.NET Could Allow Information Disclosure. [Online]. Available: http://www.microsoft.com/technet/security/bulletin/ms10-070.mspx

[22] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password Interception in a SSL/TLS channel," *Advances in Cryptology-CRYPTO 2003*, pp. 583–599, 2003.

[23] V. Klima and T. Rosa, "Side Channel Attacks On CBC Encrypted Messages in the PKCS# 7 Format," *IACR ePrint Archive*, vol. 98, p. 2003, 2003.

[24] E. Young, T. Hudson, and R. Engelschall, "OpenSSL: The Open Source Toolkit for SSL/TLS," *World Wide Web, http://www.openssl.org/, Last visited*, vol. 9, 2011.

[25] A. Dey and S. Weis, "Keyczar: A Cryptographic Toolkit," 2008.

[26] P. Gutmann, "Cryptlib Encryption Tool Kit," 2008.

[27] D. Bernstein, "Cryptography in NaCl."

[28] A. Yau, K. Paterson, and C. Mitchell, "Padding Oracle Attacks on CBC-mode Encryption with Secret and Random IVs," in *Fast Software Encryption*. Springer, 2005, pp. 299–319.

[29] S. Stubblebine and V. Gligor, "On Message Integrity in Cryptographic Protocols," in *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*. IEEE, 2002, pp. 85–104.

[30] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 180–189.

[31] S. Bellovin, "Problem Areas for the IP Security Protocols," in *Proceedings of the Sixth Usenix Unix Security Symposium*, 1996, pp. 205–214.