

Practical Padding Oracle Attacks

Juliano Rizzo*

Thai Duong[†]

May 25th, 2010

Abstract

At Eurocrypt 2002, Vaudenay introduced a powerful side-channel attack, which is called padding oracle attack, against CBC-mode encryption with PKCS#5 padding (See [6]). If there is an oracle which on receipt of a ciphertext, decrypts it and then replies to the sender whether the padding is correct or not, Vaudenay shows how to use that oracle to efficiently decrypt data without knowing the encryption key. In this paper, we turn the padding oracle attack into a new set of practical web hacking techniques. We also introduce a new technique that allows attackers to use a padding oracle to encrypt messages of any length without knowing the secret key. Finally, we show how to use that technique to mount advanced padding oracle exploits against popular web development frameworks.

1 Introduction

In this research, we show that widely used web development frameworks and web sites are using encryption incorrectly in a way that allows attackers to read and modify data that should be protected. It has been known for years in the cryptography community that encryption is not authentication. If encrypted messages are not authenticated, data integrity cannot be guaranteed which makes systems vulnerable to practical and dangerous chosen-ciphertext attacks, one of them being the padding oracle attack presented by Vaudenay at EuroCrypt 2002 (See [6]). As

explained in Paterson and Yau's summary in [5], the padding oracle attack requires an oracle which 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. The easiest fix for the padding oracle attack is to encrypt-then-MAC, i.e., encrypting information to get the ciphertext, then protecting the ciphertext integrity with a Message Authentication Code scheme. For more details on Vaudenay's attack and suggested fixes, please see [7, 1, 3, 4, 5].

In Section 2, we describe manual and automated testing techniques to find padding oracles in real life systems. In Section 3, we describe basic padding oracle attacks to crack CAPTCHA, and decrypt secret data of popular web sites and web development frameworks. In Section 4, we introduce advanced padding oracle attacks that allow attackers to mount the most interesting exploits such as creating malicious view states to run arbitrary code in JavaServer Faces. In Section 5, we describe padding oracle vulnerabilities in some other popular web development frameworks besides JavaServer Faces. We conclude in Section 6.

2 Finding Padding Oracles

If you start looking today, you will see that the padding oracle vulnerability is pervasive. It is ev-

*<http://netifera.com>, juliano@netifera.com

[†]<http://vnsecurity.net>, thaidsn@vnsecurity.net

erywhere like SQL Injection or Cross Site Scripting. This is because few people understand that attackers can decrypt their secrets if they leak out just a single bit of information. Unfortunately, the reality is that if you somehow let attackers know whether or not an error has occurred while you decrypt data supplied by them, then they can decrypt your messages.

It is also important to stress that people often use one global crypto key and a fixed IV to encrypt everything in their systems, so if attackers discover a padding oracle, they can then use it to decrypt all data encrypted under that key and IV.

2.1 Finding Potential Padding Oracles

We have been using three methods to find potential padding oracles:

Google Hacking We look for known error messages and standard API exceptions. In Java, the error message is `Given final block not properly padded`, and the standard exception is `javax.crypto.BadPaddingException`. Other platforms and crypto libraries provide different error messages and API exceptions. Just googling these messages, we promise that you can find many applications potentially vulnerable to the padding oracle attack.

Source Code Auditing Another way is to look for known source code keywords. You can start by looking for code that imports low level cryptography libraries such as OpenSSL, Crypto++, PyCrypto, Microsoft Crypto API or Java Cryptography Extension. Then look for routines that perform encryption and decryption. If there's some code to handle errors while decrypting, and/or no evidence of authentication checks, then it is highly probable that you have found a target for the padding oracle attack.

Backbox Testing We crawl the target web site to find `BASE64` strings which can be found in hidden

fields, cookies, or request parameters¹. Then we decode each `BASE64` string found. If the result looks random, and its length is a multiple of a common block cipher size, i.e., 8, 16 or 32 bytes, then there's a good chance that it is a ciphertext. We also look for common separators, i.e., `--`, `|` or `:`, which are often used to separate IV, ciphertext, or MAC. Then we replace a byte at the end of the ciphertext by a random value, then send it back to the target, and see what changes in the response. If there is an error message, then there's a high chance that this is a padding oracle. Even a blank page is enough information to perform this attack.

For blackbox testing, the most important task is to analyse and understand the meaning of error messages returned by the target upon receiving mangled ciphertexts. In short, you need to know which responses from the target mean that the padding was `VALID` after decrypting your modified ciphertexts.

This algorithm, which is very similar to the last word decryption algorithm that Vaudenay described in his seminal paper (See [6, Section 3.1]), can be used in a black-box scenario as the first step to learn if the target leaks information about its padding validity.

1. Determine the block size b . You can use the algorithm described in Section 2.2.
2. Chose a few random bytes r_1, \dots, r_b where b denotes the cipher block size in bytes, and take $i = 0$.
3. Take $r = r_1| \dots | r_{b-1} | (r_b \oplus i)$, where $|$ denotes byte concatenation.
4. Send $r|y$ to the target where y is a valid ciphertext block that you found during the manual testing phase. Record the value of i , content length, elapsed time and content type of the response. Increment i , and go back to step 3 until $i > 255$.

¹A funny side effect of doing this research is now both of us are obsessed with `BASE64` strings. Every time we see a `BASE64` string, we decode it, and if it's not plaintext, we assume it is a ciphertext, then go on trying to decrypt it using Padding Oracle attack. It's really fun!

5. Now you have 256 responses. If all of them are the same, then it is bad news, the target is not easily showing you that it is vulnerable to the padding oracle attack. Otherwise, look at each value of i where the responses are different from the rest. Manually resend each corresponding $r|y$, and examine carefully each response to see what happens.

2.2 Confirm The Existence of Padding Oracles

After doing some manual testings, one usually needs to use an automated tool to confirm the existence of padding oracles. We released POET (Padding Oracle Exploitation Tool) ² which finds and exploits padding oracles automatically.

If you want to write your own tool to detect padding oracles, you can implement the following algorithms.

Determine the block size b All padding oracle attacks need a correct block size b . The most common block sizes are 8 and 16 bytes, thus of course we can use trial and error. The following algorithm can be used to guess the correct block size by sending just one request to the target. The algorithm is not perfect but it is good enough for most cases.

1. If $\text{len}(C)\%16 = 8$, where C is some captured ciphertext, then stop and output 8.
2. Take $y = C[-16 :]$, i.e., y is the last sixteen bytes of C .
3. Send $C|y$ to the target, if it returns VALID padding then stop and output 8.
4. Output 16.

Confirm The Existence of Padding Oracles

1. Determine the block size b if possible, otherwise try the following steps for each potential block size, i.e., 8 and 16.

²Download POET at <http://netifera.com/research>

2. Use the algorithm described in [1] to find the padding length l . This takes $\log_2(b)$ oracle queries.
3. Denote the last block of a valid captured ciphertext as x_0, x_1, \dots, x_b . Take $v_i = l$, for $i = 1, \dots, l$. Run Vaudenay's Postfix Equality Check algorithm (See [6], Section 3.4) with $y_0, y_1, \dots, y_N = x_0, x_1, \dots, x_b$ and $w_0, w_1, \dots, w_m = v_0, v_1, \dots, v_l$. In other words, check to see if the last l bytes of the last block are the same as the padding bytes.
4. If Vaudenay's Postfix Equality Check returns true, then we can say that we found a padding oracle with a high level of certainty. Otherwise, we cannot be sure if the target is vulnerable or not, but we are sure that we cannot use the oracle to decrypt a ciphertext in a reliable way.

3 Basic Padding Oracle Attacks

Since HTTP is a stateless protocol, web developers must either manage states on the server, or push them to the client. For performance and scalability reasons, many web developers tend to go with the latter method. They want to keep the state as a secret, and turn to cryptography which is the right tool. However, they use it wrongly, i.e., neither apply a MAC to the ciphertext nor use an authenticated block cipher mode, and make their systems vulnerable. In this section, we show two basic padding oracles attacks.

3.1 Cracking CAPTCHA

CAPTCHA is the most popular technique to prevent computer programs from submitting automated requests to web servers. A common type of CAPTCHA requires that users enter an alphanumeric code from a distorted image. We found that some crypto-based CAPTCHA systems are the simplest examples of padding oracle attacks.

A vulnerable CAPTCHA system works as follows:

1. The server generates a random code, encrypts it using CBC-mode under some key K and some IV:

$$ERC = E_{K,IV}(rand())$$

2. This ERC will be used as a parameter for some `captcha.jsp`³ which upon receipt of a ERC , will decrypt it, and generate a distorted image. If a HTML form needs to show a CAPTCHA, it just puts something like `/captcha.jsp?token=ERC` into a `` tag to load a distorted image.
3. ERC will be stored either as a hidden field in the CAPTCHA form or as a cookie, so once a user submits a form, it will be sent back to the server.
4. Then the server goes on decrypting ERC , and compares it with the code that the user has entered. If equal, the server accepts the request; it denies the request otherwise.

Because `captcha.jsp` would decrypt any ERC sent to it, it is vulnerable to the padding oracle attack. As we discussed in Section 2, the only remaining problem now is to know when padding is `VALID`, and when it is not.

Fortunately, most CAPTCHA systems would send back an error notification when they fail to decrypt ERC , i.e., padding is `INVALID`. Some servers send either empty responses or HTML with an error message. In addition, when you modify ERC so that the padding is `VALID`, `captcha.jsp` would display an image with a broken code.

If things work out that way, attackers now have a padding oracle, and they can use it to decrypt any ERC to get its random code, hence bypass the CAPTCHA protection completely.

CAPTCHA with secret IV

$$P_0 = IV \oplus D_{PaddingOracle}(C_0)$$

³Please note that what we describe here works for any platform and language, we just use Java/JSP as an example

attackers need to know the IV to be able to get P_0 . In other words, if the IV is secret, it is not possible to decrypt the first block, making it impossible to break CAPTCHA systems whose P_0 contains part of the random code.

Fortunately, for those CAPTCHA systems that we have found during this research, the IV can be recovered easily with human intervention. Most of the time the text shown in the CAPTCHA image is exactly the same as P_0 , so if attackers know $D_{PaddingOracle}(C_0)$, then they can compute the secret IV as following:

$$IV = Human \oplus D_{PaddingOracle}(C_0)$$

where $Human$ denotes that somebody reads P_0 from the CAPTCHA image. This is very useful to attack CAPTCHA systems where manually discovering the IV a single time allows attackers to decrypt any new challenges, given the IV is not changed, without any further human intervention.

3.2 Decrypting JSF view states

JavaServer Faces introduces a powerful and flexible system for saving and restoring the state of the view between requests to the server. JSF implementations support two primary mechanisms for saving states, based on the value of the `javax.faces.STATE_SAVING_METHOD` initialization parameter. If this parameter is set to `client`, it would cause the saved state to be included in the rendered markup that is sent to the client (such as in a hidden input field for HTML). The state information must be included in the subsequent request, making it possible for JSF to restore the view without having saved information on the server side.

Although JSF specification advises that state information should be encrypted and tamper evident, as far as we know no implementation follows that advice. Some frameworks such as SUN Mojarra and Apache MyFaces do encrypt state information, but they don't protect the integrity of encrypted states which makes them vulnerable to padding oracle attacks.

By default, all JSF frameworks would display a very detailed error message if the decryption of a view state fails, which makes the padding oracle very obvious: if sees `javax.crypto.BadPaddingException`, then it is INVALID padding; it is VALID padding otherwise.

Most JSF frameworks allow developers to turn off error messages. Then attackers can use the following trick. Say an attacker wants to decrypt block C_i of an encrypted view state $C_0|C_1|\dots|C_{n-1}$, then he would append $C_{random}|C_i$ to create $C_0|C_1|\dots|C_{n-1}|C_{random}|C_i$, and send this message to the server. Since JSF frameworks ignore those extra blocks while decrypting and deserializing view states, the attacker can have a safe bet that it is VALID padding if the server returns the same page as when the view state is unaltered. It is probably INVALID padding if he sees something else, e.g., a HTTP 500 error message.

View states usually do not contain very sensitive data, but it is important to stress that some frameworks save to the client not only the view, but also the entire managed beans which could possibly contain confidential data ⁴.

4 Advanced Padding Oracle Attacks

4.1 Using Padding Oracles to Encrypt

A padding oracle is all attackers need to decrypt messages. But can it help if their goal is to encrypt messages? The short answer is yes. We designed the following technique, which allows attackers to use a padding oracle to encrypt messages of any length without knowing the secret key. Given the surprisingly fruitful consequences of this technique, we are surprised that it was not published before. We call it CBC-R encryption, and Section 4.1.2 shows that CBC-R has permitted us to mount the most interesting exploits.

⁴For Apache MyFaces, see <http://wiki.apache.org/myfaces/SaveState>

4.1.1 CBC-R Encryption

CBC-R turns a decryption oracle into an encryption oracle ⁵. CBC decryption works as following:

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

$$C_0 = IV$$

Look at the XOR operation. If attackers control C_{i-1} , and they know $D_K(C_i)$, then they can make P_i equal to anything they want. Of course attackers can control C_{i-1} because this is a chosen-ciphertext attack. For the second condition, attackers don't have access to K , but they can use a padding oracle to get $D_K(C_i)$. In other words, attackers can use a padding oracle to encrypt messages of any length without knowing the secret key.

The process is simple. Attackers choose a random ciphertext block, call it C_i . Any random block would work. They send that block to the padding oracle to get its intermediate plaintext, call this operation $D_{PaddingOracle}(C_i)$. Since

$$P_i = D_{PaddingOracle}(C_i) \oplus C_{i-1}$$

and attackers control C_{i-1} , they can make P_i equal to any block they want. Suppose they want to make P_i equal to some P_x , then all they need to do is to set:

$$C_{i-1} = P_x \oplus D_{PaddingOracle}(C_i)$$

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

$$C_{i-2} = P_{i-1} \oplus D_{PaddingOracle}(C_{i-1})$$

Now they have two consecutive plaintext blocks P_{i-1} and P_i of their choice, and a leading garbled block P_{i-2} that they can correct by inserting a new ciphertext block C_{i-3} . Repeating this operation, they can

⁵Please note that padding oracle is just one kind of decryption oracles that can work well with CBC-R

1. choose a plaintext message, and divide it into N blocks of b bytes P_0, P_1, \dots, P_{N-1} .
2. chose a few random bytes r_1, r_2, \dots, r_b , and set $C_{n-1} = r_1|r_2|\dots|r_b$
3. for $i = N - 1$ down to 1:
 $C_{i-1} = P_i \oplus D_{PaddingOracle}(C_i)$
4. $IV = P_0 \oplus D_{PaddingOracle}(C_0)$
5. output IV and $C = C_0|C_1|\dots|C_{n-1}$.

Figure 1: CBC-R pseudocode

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 attackers can set the IV , then the decrypted data will be exactly as what they want without any garbled block. If attackers don't control the IV , then the first block is garbled. In the next paragraph, we discuss what attackers can do if they don't control the IV .

CBC-R Without Controlling IV Different cryptosystems handle IV in different ways. IV can be either a prefix of the ciphertext, and totally controllable by attackers, or a fixed well known value, but attackers cannot change it. Cryptosystems also use secret IVs, then either change them every once in a while, or set them as a fixed static value.

We said that CBC-R allows attackers to encrypt any message, but if they cannot set the IV, the first plaintext block will be random and meaningless. If the victim expects the decrypted message to start with a standard header, and attackers don't control the IV, then the victim will ignore the forged message constructed by CBC-R. This is what happens with compressed data, and Java serialized object streams to name a few.

This limitation could prevent some of the highest impact attacks, and we have not found generic way to overcome it. However, we have found workarounds for particular cases.

Using Captured Ciphertext As Prefix If attackers capture a ciphertext whose plaintext is a valid message, then they can prepend the ciphertext to their CBC-R encrypted message to get a valid header after decrypting:

$$P_{valid} = D_K(C_{captured}|IV_{CBC-R}|P_{CBC-R})$$

The resulting forged plaintext message will have a valid header, but it still has a garbled block at the position of IV_{CBC-R} . This broken block can still make the victim reject the message, but we can make the victim ignore it if we choose the prefix carefully, i.e., the garbled block becomes part of some string that doesn't affect the semantic of the message such as a comment or textbox label.

Brute-Forcing C_0 In CBC-R, the final block C_{n-1} is a random block (See Figure 1). Each different C_{n-1} would yield a different C_{n-1}, \dots, C_0 chain. In other words, CBC-R can produce many different ciphertexts that are decrypted to the same plaintext block chain P_{n-1}, \dots, P_1 . The only difference is the first plaintext block which is computed as following:

$$P_0 = D_K(C_0) \oplus IV$$

Attackers want P_0 to contain a valid header. In some systems, this means that the first few bytes of P_0 must match some magic numbers. There are also systems that accept a message only if the first byte of P_0 matches the message length. If this is the case, and if the message is short enough, attackers can try their luck by brute-forcing C_0 .

Attackers change C_{n-1} , hence change C_0 , until they can get a valid P_0 . For example, if the first byte of P_0 must match the message size, trying at most 256 different CBC-R ciphertexts is enough to obtain a valid message. For longer messages or more complex validation rules, brute-forcing is not practical.

4.1.2 CBC-R Applications

Creating Malicious JSF view states It is easy to see that attackers can use CBC-R to create ma-

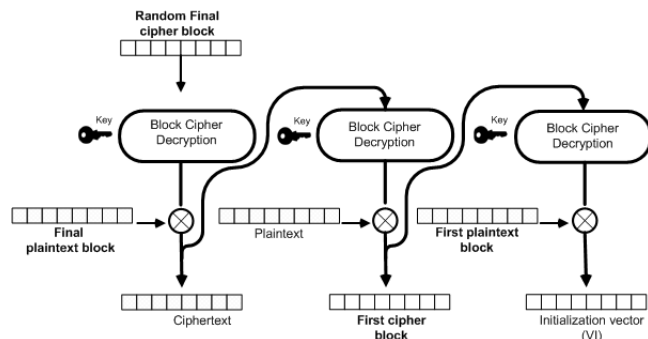


Figure 2: CBC-R Encryption

malicious view states that in the worst case could allow them to execute code on vulnerable JSF systems. The two remaining questions are:

Which view state to create? The book of Apache MyFaces and Facelets technology observed that (See [8]):

[...]The view is restored by reversing the process used to obtain the view state: it is decoded and deserialized. This poses a major security challenge to any JSF implementation because Sean has the freedom to change the view state. He can toggle the rendered attribute of UI controls that are not supposed to be available to him. He can point a `commandButton` to a method on any managed bean in the application. He can circumvent an action listener.

While we were writing this paper, a researcher published an advisory describing vulnerabilities in Apache MyFaces and SUN Mojarra, and claimed that (See [2])⁶:

⁶It is important to stress that the authors of [8] and [2] were wrong when they suggested that encrypting view states would prevent the attacks they described

[...]it is possible for an attacker to supply a new or modified view object as part of a request. The malicious view can contain arbitrary HTML code (allowing Cross-Site Scripting), and arbitrary Expression Language (EL) statements that will be executed on the server. The EL statements can be used to read data stored in user-scoped session variables, and application or server-scoped variables.

How to solve the garbled block problem? We have to solve the garbled block problem because we can't control the IV in the JSF frameworks that we have tested. The solution depends on the content of JSF view states which are Java Object Serialization Streams. The generic solution is to use the technique described in Section 4.1.1 to prepend known valid ciphertext to our CBC-R encrypted view state, and make the garbled block become part of a string that doesn't affect the semantic of the view state such as a textbox label.

Please note that although we attack only JSF view states, our techniques can be applied to exploit other kind of state information in different formats such as XML, serialized objects, JSON, simple comma separated variable-value pairs, etc.

4.2 Distributed Cross-Site Padding Oracle Attack

As we have demonstrated up to this point, all attackers need to exploit padding oracle vulnerabilities is a single bit of information. If a web site leaks out that 1-bit information, then there are a lot of ways for attackers to obtain it using cross-domain information leakage bugs in web browsers.

If you are familiar with web browser security, you probably know that JavaScript at `evil.com` can not read the response of a request to `victim.com`, otherwise this would allow all kind of abuses from evil web sites. But there's nothing to stop `evil.com` referencing resources on `victim.com`, observing how the server responds, and deducing information.

Using `` tag plus the `onerror()/onload()` events, JavaScript at `evil.com` can make web browsers to load an image at `victim.com`, and know if the image is loaded or not. This is 1-bit information, and as you know, it is enough for the padding oracle attack to work: if the image is loaded, then it is `VALID` padding; otherwise, it is `INVALID` padding.

This technique has allowed us to successfully decrypt all CAPTCHA on a target web site using only JavaScript hosted in a different server⁷. If a target is interesting enough, attackers could inject JavaScript code into popular web sites, and when people visit those web sites, the code will run in their browsers, and use their CPU time and Internet connection to decrypt the target's secrets. It is possible to distributively build a code book, i.e., a mapping of ciphertext to corresponding plaintext under the same key and IV as the padding oracle. This code book in turn can be used to automatically bypass CAPTCHA protection with 100% accuracy regardless of the graphical complexity.

5 Vulnerable Web Frameworks

Besides JavaServer Faces, we have also audited some other popular web frameworks to see if they are vulnerable to padding oracle attacks. Here are some of our findings. We will publish more results in the near future.

5.1 Ruby On Rails

Ruby On Rails, which was created in 2003, is one of the most widely used web development framework in the world. Since version 2.3, Ruby On Rails has introduced `ActiveSupport::MessageEncryptor` which is a set of functions “to provide a simple way to encrypt information for storage in an untrusted location (like cookies).” If you look at `ActiveSupport::MessageEncryptor`'s source code, you would probably see that applications that use the

⁷Watch <http://youtube.com/watch?v=e46A-PUpDvk>

provided `encrypt/decrypt` functions would be vulnerable to padding oracle attacks. It is ironic that the developers of `ActiveSupport::MessageEncryptor` do provide a secure pair of functions to `encrypt/decrypt` data that are not vulnerable to padding oracle attacks, but they still keep the vulnerable ones.⁸

5.2 OWASP ESAPI

OWASP ESAPI, which stands for OWASP Enterprise Security API Toolkits, is a project that claim to “help software developers guard against security-related design and implementation flaws.” However, we found that all OWASP ESAPI for Java up to version 2.0 RC2 are vulnerable to padding oracle attacks. There were some significant changes in ESAPI Encryption API since 2.0 RC3. Unfortunately, while these changes are heading towards the correct direction, i.e., signing the ciphertext or using an authenticated encryption mode, but at the time of this writing, there are still some bugs in the latest implementation that make applications using ESAPI for Java still vulnerable to padding oracle attacks. We leave the finding of these bugs as an exercise for readers.

6 Conclusion

In summary, in this paper we showed that padding oracle attacks allow us to decrypt ciphertext without knowing the key. We also described how to use padding oracle attacks to break CAPTCHA systems, and decrypt JSF view state. We also introduced CBC-R, a new technique that turns a decryption oracle into an encryption oracle, and allow us to create malicious JSF view states. We also demonstrated how to leverage cross-domain site information leakage in web browsers to deploy a distributed padding oracle attack that allow us to crack all CAPTCHA codes of a target web site using only Javascript hosted in a different server.

⁸Fortunately, `ActiveSupport::MessageEncryptor` is not widely used.

We hope that this research has convinced you that rolling your own crypto is extremely risky, and should be avoided. After all, the padding oracle attack has been known in crypto academic community since 2002. After 8 years, however, we still have a large number of systems vulnerable to this attack. What is even more surprising is the fact that we were the first to identify this vulnerability in popular technologies like JavaServer Faces.

Since August 2009 we have been carrying out a research in which we test-run a number of identified practical crypto attacks on random widely-used software systems. To our surprise, most, if not all, can be attacked by one or more of well-known crypto bugs. This case is just one example. We hope that publishing this vulnerability and other future results from our research would encourage the security community in taking a more serious look at crypto bugs in software system which is as pervasive as SQL Injection or XSS in early 2000.

We hope you enjoy reading this paper as much as we enjoyed writing it.

Acknowledgements We would like to thank Huong L. Nguyen, Agustin Azubel, Thomas Ptacek, rd, Gunther, Bruce Leidl, and Alex Sotirov for reading and editing the draft of this paper.

References

- [1] J. Black and H. Urtubia. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002, pages 327–338. USENIX, 2002.
- [2] D. Byrne. “Multiplatform View State Tampering Vulnerabilities”. Trustwave’s SpiderLabs. 8 Feb. 2009. Trustwave.
- [3] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. In Proc. CRYPTO 2003, D. Boneh (ed.), LNCS Vol. 2729, pp. 583–599, 2003.
- [4] V. Klima and T. Rosa. Side Channel Attacks on CBC Encrypted Messages in the PKCS#7 Format. Cryptology ePrint Archive, Report 2003/098, 2003.
- [5] K.G. Paterson and A. Yau. Padding Oracle Attacks on the ISO CBC Mode Padding Standard. In T. Okamoto, editor, Topics in Cryptology — CT-RSA 2004, volume 2964 of Lecture Notes in Computer Science, pages 305–323. Springer-Verlag, 2004.
- [6] S. Vaudenay. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS...In L. Knudsen, editor, Advances in Cryptology — EUROCRYPT 2002, volume 2332 of Lecture Notes in Computer Science, pages 534–545. Springer-Verlag, 2002.
- [7] A. K. L. Yau, K. G. Paterson, and C. J. Mitchell. Padding Oracle Attacks on CBC- Mode Encryption with Secret and Random IVs. In H. Gilbert and H. Handschuh, editors, Proceedings of FSE 2005, volume 3557 of LNCS, pages 299–319. Springer- Verlag, 2005.
- [8] Z. Wadia, M. Marinschek, Hazem Saleh, and Dennis Byrne. Antipatterns and Pitfalls. In The Definitive Guide to Apache MyFaces and Facelets, pages 229-269. Apress, 2008