Windows into the Past: Exploiting Legacy Crypto in Modern OS's Kerberos Implementation

Michal Shagam and Eyal Ronen, *Tel Aviv University*

https://www.usenix.org/conference/usenixsecurity24/presentation/shagam

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Windows into the Past: Exploiting Legacy Crypto in Modern OS's Kerberos Implementation

Michal Shagam   *Eyal Ronen*

Tel-Aviv University

Abstract

The Kerberos protocol is used by millions of users and network administrators worldwide for secure authentication, key distribution, and access control management to enterprise networks and services. Since its initial public deployment in 1989, the protocol has undergone many revisions to incorporate new cryptographic primitives and improve security. For example, initially based solely on users' passwords and symmetric cryptographic primitives, current implementations also support smartcard-based authentication with asymmetric cryptographic primitives for improved security. However, this iterative revision process has resulted in implementations riddled with legacy crypto primitives and protocol designs.

In this work, we show how we can exploit this legacy crypto to completely break the security of the enterprise network. Firstly, while arguably more secure, smartcard-based authentication uses RSA encryption with the notorious PKCS #1 v1.5 padding scheme. Although the RSA decryption is done securely inside the smartcard, a non-constant time unpadding code runs on the client's CPU. This makes both Windows's and several Linux distributions' implementations vulnerable to the Bleichenbacher attack that can recover cryptographic session tokens. Secondly, we show that the RSA smartcard-based authentication does not provide forward secrecy to the cryptographic tokens that the server provisions to the client. Thirdly, we propose and analyze different algorithmic approaches to minimize the overhead required to handle noisy oracles in the Bleichenbacher attack. This general Bleichenbacher attack analysis may be of independent interest.

Finally, we demonstrate microarchitectural side channel-based end-to-end attacks on the Windows Kerberos implementation. We start by showing how to recover tokens used to encrypt session transferred remote files by Samba. We then show how to amplify the number of decryptions performed with a single user's PIN code input, allowing us to accelerate our attack and recover users' (and admins') credentials before expiration. In addition, we describe a remote attack vector that allows us to perform the attack and generate queries.

1 Introduction

The Kerberos protocol [53, 54] is used for secure authentication and access to remote services over insecure networks. It is based on a centralized approach where a dedicated Key Distribution Center (KDC) server is responsible for user authentication and the distribution of keys. It is used across many operating systems [8, 9, 21, 24, 26, 43, 60, 71] and within many organizations [16, 20, 25] and is therefore a prime target for attacks [12, 39, 69]. Current implementations have added support for asymmetric encryption [72] and smartcard-based authentication [7].

Currently, smartcards [7] are considered a go-to secure authentication tool. Smartcards often embed *two factors* of authentication - something you have, which is the possession of the smartcard itself, and something you know, which is the smartcard pin code. The smartcard integration goal was to enhance the security of the Kerberos protocol [27], removing its reliance on users' passwords, which are a known weak point (e.g., due to users choosing weak passwords, large password breaches, phishing attacks, etc.) [17, 18, 34, 38, 49, 51, 67].

Moreover, as the private keys are stored and used only inside the secure smartcard, one can hope that the authentication process will be more resilient to attacks against the cryptographic implementations, such as Bleichenbacher's padding oracle attack [13]. However, although smartcards may securely perform the RSA decryption calculation, they still rely on the host computer to perform all the delicate handling of the PKCS #1 v1.5 unpadding and verification. Even though the Kerberos protocol does not provide a timing oracle to a network adversary, running the unpadding and verification code on the host machine may make it vulnerable to microarchitectural side-channels-based Bleichenbacher attacks.

In this work, we set out to answer the following question:

Are modern implementations of Kerberos smartcard based authentication secure against padding oracle attacks?

Table 1: Implementation Vulnerability Survey

OS	Kerberos Protocol	Smartcard Interface	Vulnerable
Windows	Security Package	basecsp	both
macOS	Heimdal	OpenSC	both
FreeBSD	Heimdal	OpenSC	both
Ubuntu	MIT	OpenSC	interface
RedHat	MIT	OpenSC	interface
Gentoo	MIT	OpenSC	interface
OpenSuse	MIT	OpenSC	interface
CentOS	MIT	OpenSC	interface
ArchLinux	MIT	OpenSC	interface
Suse	MIT	OpenCryptoki	no [a]

[a] We note that OpenCryptoki did include some non constant-time code that was identified by the tools from [33], but exploiting it using using a micro-architectural side-channel attack is challenging and would probably require a prohibitively large number of queries.

1.1 Our Contribution

Unfortunately, we discovered that all major implementations of RSA [63] smartcard-based Kerberos authentication are vulnerable to padding oracle attacks based on microarchitectural side channels. Moreover, protocols such as Samba [41] rely on Kerberos for authentication and key distribution even though Kerberos does not provide forward secrecy. This combination leads to practical attacks that allow unprivileged machine-in-the-middle (MiTM) adversaries to access restricted files in the network. Finally, by exploiting the lack of a global rate limit on the number of protocol sessions initiated in Windows, we can accelerate our attack to stealthily steal nonexpired users' authentication tokens that allow us to impersonate users and admins in the network and gain access to all of their information and capabilities.

More specifically, our contributions are as follows:

Survey of major RSA smartcard-based Kerberos implementations. We surveyed (using manual inspection of open source code and reverse engineering of close source binaries) the implementations used by the default configuration of Windows, MacOS, and several Linux distributions. The survey included all three major implementations of the Kerberos protocol — Windows' closed-source Security Package and the open-source Heimdal and MIT implementations. Only MIT implemented the required mitigations against Bleichenbacher's attack, and both other implementations include nonconstant-time code that can be exploited. We also surveyed the default smartcard interfaces used — Windows' closed-sourced basecsp, and the open-source OpenSC and OpenCryptoki. Both basecsp and the widely deployed OpenSC contain nonconstant-time code in their implementations of PKCS #1 v1.5 unpadding which make them vulnerable to Bleichenbacher's attack. The summary of our findings can be found in Table 1, where from the 10 operating systems we surveyed, 9 were found to use a vulnerable default configuration, including Windows, MacOS, and Ubuntu.

We note that as a step in the right direction, the recent Windows 11 22H2 versions disabled the use of RSA smartcard authentication in Kerberos (although the vulnerable implementations still remain). However, the currently supported and widely used Windows 10 (70% of Windows computers [57]) and Windows 11 22H1 versions remain vulnerable.

Proof-of-Concept (PoC) of Bleichenbacher Oracle on Windows. We exploited one of vulnerabilities we discovered in Windows to implement a PoC of our micro-architectural side-channel attack and create a Bleichenbacher padding oracle. We then tested and analyzed the accuracy of the oracle.

Handling Noisy Oracles in the Bleichenbacher Attack. Noisy oracles are a significant challenge to Bleichenbacher's attack. Although they are very important in practice, they have only received very limited attention in the literature [15], and the current trivial approach can double the number of required queries [64]. We experimentally explored several methods to improve the soundness of the Bleichenbacher attack and its robustness to noisy oracles. Our novel traceback approach utilizes the statistical properties of the attack. It allows us to minimize the overhead of additional required queries compared to a perfect oracle. While our approach might slightly reduce the overall success rate, it is specifically suitable for cases where we only have a limited time to perform our attack. In this case, only attacks with a small number of queries can succeed in time, so algorithms that require high query overhead to handle noisy oracles (e.g., repeating queries to improve soundness) will cause the attack to fail.

End-to-end attack implementation on Samba's file encryption. To show the practicality of our MiTM attack, we experimentally verified our ability to recover files encrypted using the SMB3 [44] protocol. The SMB3 encryption keys are sent to the client as part of the Kerberos protocol. As neither SMB3 nor Kerberos provides forward secrecy, we can exploit our attack to recover the Kerberos and SMB3 keys transferred in a previous session and recover the content of the encrypted files. This is possible even with a slow attack that may finish long after the target session expires.

Accelerated Stealthy Attack. The main bottleneck in our attack is our ability to cause the client to initiate new protocol sessions, as each initiated session results in a single padding oracle query. In many cases this requires user interaction and pin code entry, and results in only a small number of session initiation attempts that allow for padding oracle queries. This may result in a very slow attack that is potentially user detectable. In order to actively exploit the recovered session token (e.g., to impersonate an admin), our attack needs to finish in less than 10 hours before the session token and ticket expire. To overcome this limitation, we analyzed different user actions that initiate Kerberos protocol sessions. We found that specific user actions may attempt to initiate a large number of sessions (due to retries) without requiring the user's PIN code. Moreover, we discovered that although there is a limit on the number of failed attempts a single action can initiate, there is no global cap or rate limitation on the total number of protocol sessions. Finally, we show how in some settings, we can remove all requirements for user interaction by remotely initiating the requests from a malicious webpage. The attack is also stealthy as the user sees no notification, and all the resulting messages are handled by our MiTM attacker without interaction with any of the network's servers.

Open Source Kerberos MiTM Attack Framework. Together with the paper, we will release all of the code developed for our attack. This includes the full code of our MiTM framework that can manipulate Kerberos messages, and derive all of the various keys and decrypt the different messages.

1.2 Responsible Disclosure

We started a responsible disclosure process with Microsoft, the maintainers of the Heimdal Kerberos implementation, and the OpenSC maintainers. OpenSC have fully patched their PKCS #1 v1.5 code to be constant-time. We were allocated CVE-BLINDED with a high severity jointly with a disclosure from a concurrent work [33]. Microsoft acknowledged our findings as an Elevation of Privilege attack. They are currently working on a fix which they plan to release in July 2024. Although they don't have access to exact numbers, they estimate that a large number of users use smartcards for Kerberos authentication and are affected by our findings. The Heimdal team has confirmed our findings, and we are waiting for updates regarding their patching plans.

1.3 Structure of the Paper

Section 2 provides background and describes related work. Section 3 presents our threat model, describes our padding oracle, and our survey of vulnerable nonconstant-time code in Kerberos implementations. Section 4 describes our Proof-of-Concept padding oracle implementation, and Section 5 describes a full end-to-end attack on Windows. Section 6 explains how we accelerated our attack, and Section 7 discusses how to efficiently handle noisy oracles in the Bleichenbacher attack and present an active user impersonation attack. Section 8 describes how to optimize the process of finding a session that can be decrypted before it expires. Finally, Section 9 discusses the root causes for our attack and proposes various mitigations.

2 Background

2.1 Kerberos protocol

The Kerberos protocol [53, 54] is a trusted third-party authentication protocol. Initially developed in the late 1980s, it is based on the Needham-Schroeder symmetric-key protocol [52] from the 1970s, although newer implementations also support key transfer using asymmetric encryption. The protocol is widely used across operating systems [8, 9, 21, 24, 26, 43, 60, 71] and has several widely used implementations including: MIT [47], Heimdal [29, 30], and Microsoft [43]. The Kerberos protocol defines a protocol between three types of entities:

1. Client or principal — this includes the host computer itself and the user
2. Key Distribution Center (KDC) — The third-party trusted server (also known as the domain server) that provides the authentication service.
3. Application service server (Application Server) — A server that provides access to the network resources such as remote files, remote powershell and email server.

We note that some of the Kerberos entities, such as the KDC and the Application Server, are often implemented on the same physical computer.

2.1.1 Tokens in Kerberos

Kerberos uses tokens (which are called tickets) which are provided to the client during authentication, and can be used by the client to prove their identity and permissions to application servers. The tokens contain encrypted information that allows servers in the network to verify the validity of the tokens. The tokens can be used for a limited amount of time, known as a lifetime. After the token expires, the client needs to repeat the authentication process. The default token lifetime on Windows is 10 hours, this can be extended for up to 7 days with a process called ticket renewal [45].

The two main types of tokens used in Kerberos are called Ticket-granting-tickets (TGTs) and Ticket-granting-service (TGSs) sent in the Kerberos protocol messages AS-REP and TGS-REP respectively. The TGT ticket is used by the client to request TGS tickets. The TGS service ticket is then used by the client to gain access to different services on the network.

2.1.2 PKINIT

PKINIT is an extension of the Kerberos version 5 protocol [70] adapting it to support asymmetric encryption and allowing the authentication of clients without relying on (potentially weak or compromised) user passwords. The asymmetric encryption in PKINIT is used to transfer a symmetric key which we refer to as the session token. This session token is then used to derive the rest of the keys used in the remainder of the protocol.

2.1.3 Open Source Kerberos Implementations

The Microsoft Windows implementation of Kerberos is closed-source. The most common open source implementations of Kerberos, which can be used on additional operating systems, are the MIT KRB5 [8, 9, 16, 21, 24, 26, 47, 55, 60, 61, 71] implementation and the Heimdal [23, 29, 65] implementation. The Heimdal [29] Kerberos implementation, in addition to being open-source, was created as an implementation independent of US regulations and export restrictions. Therefore Heimdal [48] was historically used when export restrictions hindered the use of the MIT implementation.

2.1.4 Microsoft Active Directory

For Windows Servers, Active Directory (AD) [42] implements a domain Server, acting as the trusted party KDC, providing authentication and network resources and services including a centralized remote filesystem, remote powershell and outlook email service. The remote filesystem is most commonly implemented using the Samba [41, 44] protocol that relies on the Kerberos protocol for authentication (although some legacy authentication protocols may still be supported).

2.2 Smartcards and Multifactor Authentication

Smartcards [7] are physical devices with integrated chips that can securely store cryptographic certificates used for identification and authentication of users. To prevent the compromise of the private keys, smartcards can usually use them internally to perform cryptographic operations on behalf of the user.

In general, three common factors can be used for authentication: something you know, something you have, and something you are. Multifactor authentication is the use of two or more authentication factors. Possession of the smartcard provides the something you have factor. Smartcards are commonly used with a personal identification number (PIN) limiting the amount of incorrect attempts in order to prevent brute-force attacks which covers the something you know factor as well. Smartcards can also be incorporated with a fingerprint reader that also require biometric authentication (something you are) to use the smartcard.

2.2.1 Kerberos RSA Smartcard-based Authentication

Users can authenticate and log on to a Client using a password or alternatively with a smartcard [7]. When a user logs in or unlocks a Client, they connect the smartcard to the Client, and then the user is prompted for their pin code. The smartcard holds a private key and a public certificate that correspond to the user, and if the PIN is correct, the certificate is sent to the Client who verifies the certificate and username correspond. The client then initiates the Kerberos [27] authentication protocol, while requesting the smartcard to perform the asymmetric cryptographic operations (e.g., RSA decryption and signing) on its behalf.

The Client initiates the Kerberos authentication protocol by sending an AS-REQ message which is a request for a ticket granting ticket. The message includes the user's certificate and is signed by the smartcard.

The Server responds by sending the Client an AS-REP packet that includes the TGT. The AS-REP message includes an RSA ciphertext that encrypts a symmetric key that is used to decrypt the session token allowing the encryption and decryption of the current packet and the rest of the session. This RSA ciphertext is sent to the smartcard for decryption, and it is the main target of our attack. A more detailed description of the key scheme can be found in the protocol technical specification [70].

2.3 Padding Oracle Attacks

Padding schemes are used to encode messages before encryption. After decryption, the resulted plaintext is "unpadded" to recover the original message. The unpadding process usually involves verification of the padding. Code that implements this unpadding and verification process may leak some information about whether the decrypted padded message conforms to a specific padding structure. Attackers can use side-channel attacks to learn this information and exploit it to mount adaptive chosen ciphertext attacks that can recover the plaintext value. Side-channels can include indicative error messages, timing variations, memory access patterns, etc.

2.3.1 RSA padding

Padding schemes are used in many implementations of RSA in order to add semantical security and prevent malleability. We will describe the padding standard PKCS#1 V1.5 [31, 50] which is used for RSA encryption and is vulnerable to chosen ciphertext attacks such as the Bleichenbacher attack [13].

Let (N,e) be an RSA public key, let (N,d) be the corresponding private key, and let ℓ be the length of N (in bytes). The encryption and padding of a message m containing $k \leq \ell - 11$ bytes is performed as follows:

1. First, a random nonzero padding string PS of byte-length $\ell - 3 - k \geq 8$ is chosen.
2. Set m^* to be $\texttt{0x00}||\texttt{0x02}||PS||\texttt{0x00}||m$. Note that the length of m^* is exactly ℓ bytes.
3. Interpret m^* as an integer $0 < m^* < N$ and compute the ciphertext $c = m^{*e} \bmod N$.

The decryption routine computes $m' = c^d \bmod N$ and parses m' as a byte string. It then checks whether m' is padding standard conforming:
$m' = \texttt{0x00}||\texttt{0x02}||PS''||\texttt{0x00}||m''$ where PS'' is a string consisting of at least 8 nonzero bytes. If the message is found to be conforming the decryption routine returns m'' (which will

be equal to $m*$ if the ciphertext was not modified). Otherwise the decryption routine fails.

2.3.2 Bleichenbacher's Attack on PKCS#1 v1.5 Padding

Appendix A provides a high level description of the Bleichenbacher [13] chosen ciphertext padding oracle attack on the padding scheme defined above. The attack allows an attacker to compute a private key operation without knowing the secret private key. The attack relies on the RSA homomorphic property of multiplication.

Attack Prerequisites Bleichenbacher's attack assumes the existence of an oracle Bl which given an RSA ciphertext c as input answers whether c decrypts to a PKCS #1 v1.5 conforming message. More formally, let (N,d) be an RSA private key. The oracle Bl performs the following for every ciphertext c

$$\mathsf{Bl}(c) = \begin{cases} 1 \text{ if } c^d \bmod N \text{ has valid PKCS \#1 v1.5 padding} \\ 0 \text{ otherwise} \end{cases}$$

2.3.3 Bleichenbacher Oracle Notation

In this paper, we follow previous work [11, 64] and consider three different checks that can be performed by an implementation when validating the RSA-decrypted padded plaintext. All of the implementations we surveyed checked that the padded plaintext indeed begins with the expected value of $0x0002$. This check was followed by some combinations of the following checks:

First Check Verifies there is at least one zero byte within the padded plaintext.

Second Check Validates the first 8 bytes of the padding string are nonzero.

Third Check Verifies the length of the unpadded data.

We denote an oracle that performs all three checks as FFF and the most permissive oracle that doesn't perform all three as TTT. An ignored check is denoted with a T.

2.4 Flush+Reload Cache Attack

Microarchitectural side-channel attacks, such as cache side-channel attacks, exploit information leakage from the hardware infrastructure itself. The cache is often a shared resource between users and can be used as a side channel to learn information on another process without authorization or Administrative privileges. Attackers can use this side channel to bypass security mitigations applied on virtual memory since the cache side channel relies on the physical addresses.

Flush+Reload [74] is a side channel technique that allows monitoring of access to memory lines in shared pages. The monitored memory lines are flushed from memory and later the memory line is reloaded. The amount of cycles it takes for the memory line to be reloaded can indicate whether the memory line was in the cache. The attack can be used to target the last level cache, L3, allowing monitoring when the victim and the attacker do not share the same execution core.

2.5 SMB3

The Samba protocol [41] allows the authorized transfer of remote resources, shares, between computers on a network. In previous versions of the Samba protocol, remote files and large resources were transfered as plaintext and an attacker could recover the unencrypted files sent over the network by simply eavesdropping and reconstructing the files.

SMB3 [44] introduced the encrypted transfer option for remote resources. The principal is authenticated and the files are sent via an encrypted session. The encrypted session is created as a Kerberos based encrypted session. The Kerberos session is established and used to send the encrypted SMB3 session token. Therefore if an attacker can decrypt the Kerberos session, they can extract the SMB3 session token and reconstruct the files sent over the encrypted session *thus exploiting the lack of forward secrecy.*

2.6 Related Work

2.6.1 Attacks on Kerberos

Since the Kerberos protocol is used across many operating systems for secure authentication and managing access credentials, it is a prime target for many attacks such as Kerberoasting [39], pass the ticket attacks [69], pass the hash attacks [12], and golden ticket attacks.

There have been several credential dumping attacks on compromised hosts [19]. The attacker gains read access to the memory of the lsass process, Local Security Authority Subsystem Service, and extracts the stored credentials from memory. Microsoft has since implemented several layers of encryption and isolation techniques [46].

A recent related attack on legacy crypto in Kerberos from Google Project Zero [22] showed that the use of legacy cryptography and lack of authentication of parts of the ticket request message allows an attacker to replace the requested encryption key types with a 40-bit RC4 [59] key allowing a bruteforce attack. This allows the adversary to recover and exploit the ticket returned in the response to the request service ticket message.

2.6.2 Attacks on RSA PKCS #1 v1.5

Since the original Bleichenbacher paper [13] there has been a long line of work attacking RSA PKCS #1 v1.5 [11, 28, 36, 40, 73]. For example, the DROWN attack [10] a cross protocol MiTM attack exploiting SSLv2 [58] protocol flaws allowing an attacker to decrypt TLS [59] connections using the RSA key exchange when the same private key was used for both TLS and SSLv2. The Return Of Bleichenbacher's Oracle Threat, ROBOT [14], a chosen ciphertext attack using different server responses based on whether the padding is conforming, a padding oracle, to sign a message with the private key. The 9 Lives of Bleichenbacher's Cat [64] also exploits microarchitectural side channels and downgrade attacks for several TLS [59] implementations.

In an independent concurrent work, Kario [33] surveyed open-source implementations of PKCS #1 v1.5 decryption using an automated testing approach and have also found that OpenSC has vulnerable non constant-time code. Using their automated tool, OpenCryptoki was also shown to have non constant-time code.

3 Padding Oracles in Kerberos PKINIT

We have conducted a systematic analysis of the security of all major RSA smartcard-based Kerberos implementations in respect to Bleichenbacher's padding oracle attack. In this section we will describe the threat model we assume for our attack and the parts of the protocol we analysed. We will then present some examples of the different types of oracles we found.

3.1 Timing Base Padding Oracles

Unlike the TLS protocol [62], Kerberos does not seem to provide a network adversary with a timing oracle. As explained in section 2.2.1, the RSA ciphertext encrypts a symmetric key that is used to decrypt an authenticated ciphertext included in the same message. If a modified ciphertext results in a plaintext with incorrect padding, the Kerberos protocol will simply discard the message. Moreover, even if a modification of the ciphertext results in a plaintext with correct padding, the resulting symmetric key will be changed, and the symmetric authenticated decryption will fail. This will also cause the Kerberos protocol to discard the message.

Although the amount of time until the message is discarded differs in both cases, there is no externally visible difference in the response, as the client will simply not respond. Moreover, from our experiment, it seems that on Windows, the TCP connection used for the attack is closed independently of the type of error, preventing an oracle based on a lower communication layer such as the ROBOT attack [14]. To conclude, in our experiments, we were not able to find any timing-based oracles. However, as we will show, the code that runs on the client may still be vulnerable to micro-architectural side-channel attacks, which can still allow us to build a padding oracle.

3.2 Threat Model

Our threat model is depicted in Figure 1 and is derived from the Kerberos' main goal of allowing authentication over an insecure network. Thus we assume that all parties that participate in the protocol are honest and not compromised. This includes the servers (i.e., the KDC and various applications service servers), clients, users and smartcards. The users use smartcards to authenticate their clients to the KDC using Kerberos and for the provisioning of tokens and cryptographic keys for secure authenticated communication with the various application service servers (e.g., securely accessing remote files). The goal of our attack is to gain unauthorized access to network resources.

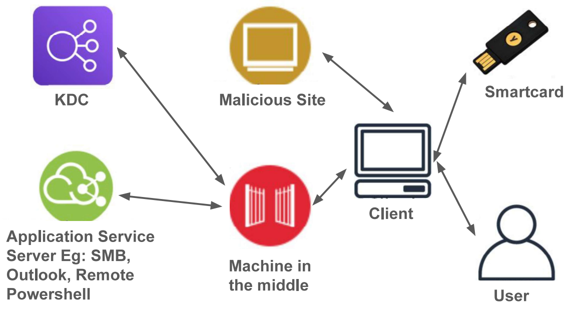
KDC
Malicious Site
Smartcard
Client
Application Service Server Eg: SMB, Outlook, Remote Powershell
Machine in the middle
User

Figure 1: Threat Model Overview - we assume a network with honest servers and an honest user trying to log in using an uncompromised smartcard and client machine. A malicious MiTM is able to intercept and modify packets over the network and to communicate with an *unprivileged* malicious program running on the Client.

As we assume that the network itself is insecure, a MiTM attacker can sniff, intercept, and modify packets sent between the client and the servers (e.g. a compromised router). To allow microarchitectural side channel attacks, we further assume that the honest user's computer, the Client, also runs *unprivileged* malicious code. The code can be an *unprivileged* program that the honest user ran, a program currently running under a *different* user account that is logged on to the same computer, assuming the other user may be compromised or deceived into running our code. Moreover, previous work shows that JavaScript code from a malicious website running inside the user's browser may perform such micro-architectural attacks [35, 37, 56, 66, 68]. Note that for our experiments, we used unprivileged native code running under a different user account. Finally, for our accelerated attack (described in Section 6.2), we further assume that the honest targeted user accesses a malicious website.

Note that we assume that the communication between the client and the smartcard is secure, and that the PIN codes are not compromised.

3.3 Oracle Description

As described in section 2.2.1, our attack focuses on the client's RSA decryption that occurs while handling the Kerberos AS-REP message, specifically the PK-AS-REP message which is part of the Kerberos PKINIT extension. RSA is used as a key encapsulation mechanism (KEM) to encrypt symmetric keys that are required to decrypt the rest of the message and derive the Kerberos session token which we want to recover in our attack.

Following The 9 Lives of Bleichenbacher's Cat [64] paper, we can partition the RSA decryption process in Kerberos to three main stages:

1. **Mathematical Calculation and Data Conversion** — First, the RSA ciphertext is decrypted and the resulting plaintext is converted into a byte array. In our case, this stage runs on the smartcard.
2. **PKCS #1 v1.5 Verification** - The byte array is then checked for conformity to the PKCS #1 v1.5 padding scheme and the padding is removed.
3. **Protocol Level** – If the byte array passes the verification stage, the Kerberos protocol will try to use the resulting plaintext as a 3DES symmetric key. If the resulting plaintext size is equal to the expected key length or padding oracle mitigations were implemented, the resulting key will be used to decrypt the rest of the AS-REP message.

In our attack we need to overcome two main challenges: The first is that we don't have direct access to the smartcard, and thus can't exploit padding oracles in the first stage of the decryption. The second and more significant challenge is that the client does not respond to any failure in the AS-REP decryption process. As the symmetric encryption part of the AS-REP is authenticated, even if PKCS #1 v1.5 validation of the modified RSA plaintext passes, the symmetric decryption will fail and the packet will simply be dropped and ignored. To overcome theses challenges, we searched for nonconstant-time code at the PKCS #1 v1.5 verification and Protocol level stages, that can be targeted using microarchitectural side-channel attacks.

3.4 Nonconstant Time Implementations

We surveyed all three major implementations of the Kerberos protocol — Windows' close-source Security Package and the open-source Heimdal and MIT implementations for nonconstant time code. While MIT properly implemented mitigations against Bleichenbacher's attack, both other implementations include nonconstant-time code that can be exploited for a padding oracle attack. We also surveyed the default smartcard interfaces used — Windows' close-sourced basecsp, and the open-source OpenSC and OpenCryptoki. We found that the most widely used interfaces, basecp and OpenSC include nonconstant-time code. Table 1 shows which packages are uses by default in 10 OSs we surveyed, and that 9 of them

```
1  int VerifyPKCS2Padding(inBuffer, inLen, out){
2  ...
3  if ((*(char *)(inLen - 2 + inBuffer)=='\x02')
4    && (*(char *)(inLen - 1 + inBuffer)=='\0')){
5    pcur = (char *)(inLen + inBuffer);
6    do {
7      if (*pcur == '\0') break;
8      pcur -= 1;
9      inLen -= 1;
10   } while (inLen != 0);
11   int i = 0;
12   if (inLen != 0) {
13     //Allocate output buffer
14     //copy plaintext to output
15     ...
16       return SUCCESS;
17     }
18     ... }
19   }
20   return ERROR;
21 }
```

Listing 1: Simplified Decompiled VerifyPKCS2Padding

use nonconstant-time code, and Table 2 shows the specific oracles we found in each package. We will now describe several examples of nonconstant-time code in Windows which is arguably the most widely used Kerberos implementation. For examples of nonconstant-time decryption in other OSs, see the full version of the paper.

3.4.1 PKCS #1 v1.5 Verification Oracles

While the smartcard performs the RSA decryption itself, the padding removal and verification is performed on the Client and implemented in the basecsp smartcard interface which has two vulnerable functions. Listing 1 shows a simplified code of the function **VerifyPKCS2Padding**. It verifies the padding is correct by verifying the first two bytes are `\x00\x02` and that the message contains an additional `\x00` byte to indicate the beginning of the data. The code lines 5 to 18, which search for an additional zero byte, are only performed when the first two padding bytes are correct leading to a TTT type padding oracle. Moreover, the code lines 13 to 16, which copies the buffer, only run if a `\x00` byte was detected, targeting these code lines leads to a different FTT padding oracle.

Listing 2 shows a simplified code of the function **CSPUnpadData**. It calls the **VerifyPKCS2Padding** function on line 3. Only if the padding check and removal is successful then the copy and swap buffer code in lines 5 to 8 runs, leading to an FTT padding oracle.

3.4.2 Protocol Level Oracles

Windows' Security Package does not implement any of the standard mitigations against Bleichenbacher attacks. Listing 3 shows a simplified code of the function **CPImportKey**. The call to function **FIsLegalKeySize** at line 6 is only made if

Table 2: Padding Oracle Summary

Oracle Type	Implementation	Function Name	Nonconstant time code
FTT	Security Package	CPImportKey	Conditional check for plaintext length
FTF	Security Package	CPImportKey	Conditional call to 3DES decryption
TTT	Basecsp	VerifyPKCS2Padding	Conditional search for zero byte
FTT	Basecsp	VerifyPKCS2Padding	Conditional buffer copy
FTT	Basecsp	CSPUnpadData	Conditional buffer copy
FFT	Heimdal	p11_rsa_private_decrypt	Nonconstant time call to interface
FTT	Heimdal	hx509_cms_unenvelope	Conditional check for plaintext length
FTF	Heimdal	hx509_cms_unenvelope	Conditional call to 3DES decryption
FFT	OpenSC	pkcs15_prkey_decrypt	Conditional buffer copy
FFT	OpenSC	sc_pkcs1_strip_02_padding	Conditional buffer copy
-	MIT	pkcs7_decrypt	-
-	OpenCryptoki	-	-

```
int CSPUnpadData(inBufferCtx, outLen,  out){
  ...
  result = VerifyPKCS2Padding(...);
  if (result == 0) {
  //Swap byte order in plaintext
  //and copy to outBuf
    ...
    *out = outBuf;  }
  ...
  return result;
}
```

Listing 2: Simplified Decompiled CSPUnpadData

```
1  int CPImportKey(csInfo){
2    ...
3    int result = CspUnpadData(csInfo,&len,&out);
4    if (result == SUCCESS){
5      ...
6      int isLegal = FIsLegalKeySize(len,out);
7      if (isLegal){
8        Decrypt_3DES(encData,out,len,&decData);
9        ...
10     }
11   }
12   return SUCCESS;
13 }
```

Listing 3: Simplified Decompiled CPImportKey

function **CspUnpadData** is successful, leading to a FTT oracle. Moreover, the call for 3DES decryption at line 8 only occurs if the resulting plaintext size corresponds to a legal 3DES key size, this leads to a stricter FTF type oracle.

4 Padding Oracle Proof-of-Concept

In this section we will describe our padding oracle PoC on Windows. As described in Section 3, we had the choice of several different oracles to target with a microarchitectural side-channel attack. Such attacks can be relatively noisy, while the Bleichenbacher attack requires a large number of queries and can be very sensitive to errors. This means that we need to take care when we choose our oracle and calibrate our attack.

The oracle which is arguably the easiest to target is the conditional call to 3DES decryption in **CPImportKey** as we can target the actual 3DES decryption process (e.g., access to the 3DES SBOXs tables in memory). As the decryption takes a relatively long time and includes a very large number of repeated memory access, it results in an attack with very high accuracy. However, this is a relatively strict FTF padding oracle as it requires the first zero byte after the padding to be at the specific location corresponding to an accepted 3DES key size. This requirement decreases the probability of getting a conforming plaintext by a factor of 2^{-8}, and significantly increases the total number of queries required by the full attack.

The most permissive oracle that requires the smallest number of total queries is the TTT oracle in **VerifyPKCS2Padding**. However, we found that, both the TTT and FTT oracles in **VerifyPKCS2Padding** are compiled to very short assembly code that co-resides in the same cachelines as other code segments that are not part of the oracles. This resulted in a very low accuracy that was not suitable for our needs.

Finally, we decided to exploit the FTT padding oracle in **CPImportKey**. Although it is stricter, the probability of conforming to it decreases by less than a factor of 2 compared to the more permissive TTT oracle. On the other hand, it is much easier to target with a microarchitectural side-channel attack leading to a much higher accuracy of our attack.

4.1 Experimental Oracle Calibration

In order to verify the existence of the padding oracle on Windows, we created a setup with a Windows domain Server, Windows Server 2019 running Active Directory and a Client

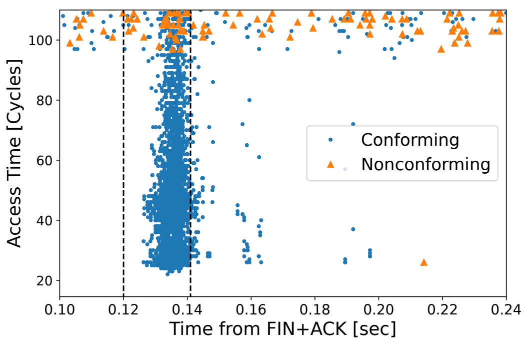

Figure 2: Calibration Hit and Access Time Cycle Count Comparison - A comparison of cache hit patterns of all calibration hits for both message types synchronized using the FIN+ACK packet time reference frame. The time range chosen, between 0.12 and 0.14 seconds after FIN+ACK, is shown by the dashed black lines.

running a Windows 10 operating system with users that can login using smartcards. The setup Client has a dual-core CPU, Core i7-7600U processor. We used a YubiKey USB token based smartcard with 2048-bit RSA key.

We implemented a MiTM attacker (that runs on a separate computer on the same LAN) that can receive AS-REQ messages from the client and respond with AS-REP messages with a modified RSA ciphertext. We implemented *unpriviliged* attack code that runs on the client and uses the Flush+Reload [74] attack to monitor the cacheline that stores the code of the oracle we exploit. Note that the native attack code runs under a different user account than the one we target. The attack code starts its measurement after receiving a command from the MiTM machine and stops after receiving another command from the MiTM machine with a specified delay. The attack code then sends back a trace of all cache hits (where the access time is below a fixed threshold). We tested this setup by sending 4000 AS-REP messages to the client, with interleaved conforming and nonconforming RSA padding. Note that we don't assume clock synchronization between the MiTM attacker and the client, and that network packet handling and processing is asynchronous and may vary in time. We used the network packets to synchronize our attack and align the different traces. We synchronized our attack to the time where the MiTM attacker receives a FIN-ACK packet from the client that confirms the AS-REP message was received.[1]

Figure 2 shows the distribution of the resulting cache hits for messages with conforming and nonconforming RSA padding aligned using the time the FIN+ACK message was received by the MiTM attacker. Although we receive a large

[1]This TCP FIN+ACK messages are always sent by the network stack and is unrelated to the decryption process. After receiving the FIN+ACK message, the MiTM attacker sends a notification messages to the attack code running on the client machine which is then used to sync the resulting trace.

Table 3: Hit Count Message Distribution

Amount	**0**	**1**	**2**	**3**	**4**
conf.	0.07	0.12	0.36	0.36	0.09
nonconf.	0.9984	0.0013	10^{-4}	$3 \cdot 10^{-4}$	$5 \cdot 10^{-5}$

number of hits also for nonconforming messages, by using the alignment and restricting ourselves to the specific time range of 0.12 to 0.14 seconds after the FIN+ACK message, we can get a relatively robust signal. Table 3 shows the distribution of hit counts within the above range for single messages. The distribution was calculated from an extensive experiment with ≈ 27000 conforming messages and ≈ 66700 non-conforming messages. Note that we still get some conforming messages with zero hits, and nonconforming messages with one hit.

Note that the resulting oracle is still too noisy to be used for a full attack. A straightforward solution is to follow the approach of Ronen et al. [64] and repeat measurements to improve the oracle's accuracy at the cost of a significant overhead in the number of oracle queries required to finish the attack. In the next section, we will assume a reliable padding oracle based on repetitions. In Section 7, we will delve deeper into this problem of noisy oracles and describe and analyze different approaches to handle them more efficiently.

5 End-to-end Attack

We will now describe how we can mount a full end-to-end attack on Kerberos. After validating the existence of a padding oracle, the main remaining challenge for our attack is the requirement for user interaction. The Kerberos authentication protocol can only be initiated by the client as a result of some user action, for example, an attempt by the user to login to the client machine. Such a login attempt will cause the client to initiate a protocol session and to send an AS-REQ message.

In the first stage of the attack, our MiTM attacker can forward the AS-REQ message to the KDC and receive from it the resulting AS-REP message, that includes the RSA ciphertext we want to decrypt (which will allow us to recover the session token). The MiTM can now use the Bleichenbacher attack algorithm to maliciously modify the RSA ciphertext and use the oracle we presented in Section 4. As described in Section 3, due to the validation of the symmetric decryption process, the protocol session will fail even if the RSA plaintext passes the padding verification.

After this first stage, there is no more interaction with the KDC. Each future protocol session initiated by the user will cause the client to send a new AS-REQ message. The MiTM will use the same AS-REP messaged he received from the KDC in the first stage, and will modify the RSA ciphertext according to the Bleichenbacher attack algorithm depending on the previous result of the oracle query.

After a login attempt, if a protocol session fails, the client will retry to initiate the protocol up to 3 times. If all 3 attempts fail, the user might be alerted that an error has occurred.[2] This means that we can use a single login attempt for up to 3 padding Oracle queries or 2 if we want to let the protocol finish successfully on the last attempt and ensure the stealthiness of the attack. Assuming a full Bleichenbacher attack will require more than 10000 queries, we require more than 5000 login attempts by the user before we are able to decrypt a single session ticket, which could take a very long time period. As the session ticket expires after about 10 hours, it will not be usable anymore.

5.1 Attack on Samba's file encryption

In this attack scenario, the MiTM has monitored and saved packets from a previous or ongoing Kerberos session where the User accessed an application service and data the User was authorized to access was sent over the Kerberos based encrypted session. We specifically target the SMB3 [41] protocol where the User accesses files on a remote filesystem and shared files are transferred over an encrypted session. The goal of the SMB3 protocol is to encrypt and protect the transfer of the remote files so that only authenticated and authorized users can access them. We exploit the fact that neither Kerberos nor SMB3 provide forward secrecy in respect to the Kerberos' long-term RSA keys and encryption. I.e., if we can compromise the RSA keys, or break a specific RSA ciphertext, we can decrypt and recover the sessions' token. Using the session token and the transcript of the protocol we can derive all the encryption keys used in the resulting SMB3 session to decrypt the files sent over the network. Although the corresponding session tokens usually expire after 10 hours by default on Windows, the same long-term RSA keys stored on the smartcard can be used for years without change. This means that as long as our MiTM can record the protocol transcript of a specific session and the SMB3 encrypted files, it can run the attack as long as the user doesn't rotate the long-term keys on the smartcard. In the end of the attack, the recovered session ticket will enable the attacker to decrypt the files.

We have implemented a full end-to-end attack based on the padding oracle described in section 4. We will open source all the code used for the attack, including the full code for the Kerberos and SMB3 key derivation and file decryption. Note that to speed up our experiments, we used the acceleration technique described in Section 6.

[2]E.g., the user might notice some resources are not available.

6 Attack Acceleration

The attack described in Section 5 can take a very long time to finish, and it only allows for a passive decryption of data sent in the past. In this section, we will describe how we can significantly accelerate our attack. In Section 7 we will show how to efficiently use the resulting noisy oracle to allow us to break Kerberos sessions before they expire, and use the recovered tokens to impersonate users in the network.

6.1 Initiation of Kerberos Sessions

As we mentioned before, our attack relies on the client machine attempting to initiate Kerberos sessions. After a user attempts to login and inputs their PIN code, windows will initiate a new Kerberos session, and if it fails, it will retry up to a total of 3 attempts without requesting the user to reenter their PIN code (thus handling temporary connectivity issues). The amount of authentication attempts is part of the domain group policy and can be configured depending on the network requirements (with default value of 3).

The logon process occurs infrequently and can take up to several minutes resulting in a very low query rate. However, we discovered that other user actions can lead to a much higher number of protocol initiation attempts. Moreover, some of them do not require the user to reenter their PIN code. For example, a user attempts to access remote files will initiate a Kerberos authentication attempt without requiring any additional user input.

File permissions are checked after the Kerberos session is established, therefore Kerberos session establishment attempts are generated before permissions are checked. For this reason, the domain user isn't required to have permission to access the shared folder in order for the vector to work.

Although the number of authentication attempts is limited by the group policy, the actual number of attempts per user action is significantly higher. This is presumably because a failure in the authentication, will cause a failure in a higher level code, that will cause it to also retry. Moreover, there is no global limitation on the total number or rate of attempts! The actions can therefore be repeated in order to initiate a large number of sessions at a very fast rate.

Table 4 shows a comparison of the amount of queries generated by different user actions and whether the actions require the PIN input. Generating the authentication requests can be as simple as opening file explorer with one of the recently used files being a file or folder that requires user authentication to access or accessing one of these files in notepad or in the run app.

The rate of generated AS-REQs varies greatly between these sources due to the additional overheads (e.g., user interaction). While runas generates about 4 or 5 queries per minute, the additional actions that do not require reentering the PIN are significantly faster and can reach over 50 queries

Table 4: Action Query Generation Comparison

Process	Session Retries	PIN required
login	~3	yes
unlock	~3	yes
First credential use	~80	yes
runas	~2	yes
file explorer	~20	no
more	~2	no
cat	~20	no
remote file access in run app	~70	no
remote file access in notepad	~80	no

```
1 <html>
2   <head>
3     <meta http-equiv="refresh" content="4">
4   </head>
5   <iframe src="file://win-r3f0hi0ntca\mysecret\
      classifiedsecrets.txt">
6   </iframe>
7 </html>
```

Listing 4: Crafted HTML Example

per minute.

6.2 Remote Access Vector

The actions we surveyed can significantly increase the amount of queries per PIN code entry, however the attack is still highly dependent on the honest user actions and the rate of actions. In order to remove this dependency, we found a remote vector which allows a malicious website to trigger the authentication process without any user interaction, by attempting to access remote files.

As described above, remote file access attempts can trigger the authentication process and generate AS-REQ ticket requests. In contrast to our micro-architectural attack code that can run under a *different* user account, only code running under our target user can trigger these accesses. As we don't want to assume that the target user will run our code, we show how we can trigger these accesses by only requiring the target user to access a malicious website. There are several methods a website can cause the browser to attempt to access remote files, e.g., using an image input, iframe input, or links. In addition, the crafted HTML file can specify an automatic refresh, which causes the browsers to keep sending requests. The example HTML code in Listing 4 demonstrates one of the methods to generate requests. Line 5 accesses a remote file which generates requests and line 3 automatically reloads the html site which causes line 5 to be repeated continuously.

Although these remote file access attempts will fail due to our attack (and, in fact, the files might not exist at all), the attack can remain stealthy since there are many ways to hide the file access errors in the browser. For example, loading an image or an iframe to an unseen location or a very small area on the screen. There is also the option of displaying alternative text when loading fails. This means that the user gets no notification of any error and remains unaware of our stealthy attack. Moreover, the attack can continue running as long as the malicious website's tab is opened. The attack will continue even when the targeted user views another tab, uses another application instead of the browser, or even locks the computer or switches to another Windows user.

In order to determine the practicality of this vector, the three most common browsers were inspected: Google Chrome, Edge (also Chromium based) and Microsoft Internet Explorer. Our attack works on all browsers for HTML files that are locally stored on the machine. The behavior for HTML files retrieved directly from remote websites is different due to different mitigations implemented in order to secure file access. While all three browsers generated queries for local HTML files, Edge and Chrome, both based on Chromium, blocked these attempts for the remote HTML sites. Thus, our attack will work on all browsers if we can cause our user to open a locally stored malicious HTML file (e.g., after downloading it from a malicious website and saving it locally). In addition, even for the case of remote HTML files, there is a vast amount of recent vulnerabilities published [1–6] that can be exploited to implement this remote vector. We thus argue that our remote attack vector is practical as, with high probability, many more similar vulnerabilities still remain.

Even after the acceleration, the overhead required to handle the noisy oracle might prevent us from finishing our attack before the Kerberos session expires. In the next section, we will describe how to reduce the overhead and perform a full end-to-end attack to hijack Kerberos sessions.

7 Bleichenbacher Attack with Noisy Oracles

As was described by Capol [15], the literature on Bleichenbacher attacks mostly focused on perfect oracles without any noise. However, in practice, many oracles are noisy. Ronen et al. [64] handled a noisy oracle by simply repeating each "positive" query multiple times to reduce the false positive rate to a negligible level. This approach is based on the intuition that the attack algorithm can recover from a false negative with some overhead, but a false positive might cause the attack to fail. As the overhead of repeating each false query is larger than the overhead of false negatives, only positive queries are repeated.

The general idea suggested by Ronen et al. [64] is to reduce the false positive rate to a negligible level (false positives should never occur) while paying the price in query overhead due to both repeated queries and the increased false negative rate. We argue that this approach is sub-optimal in both their

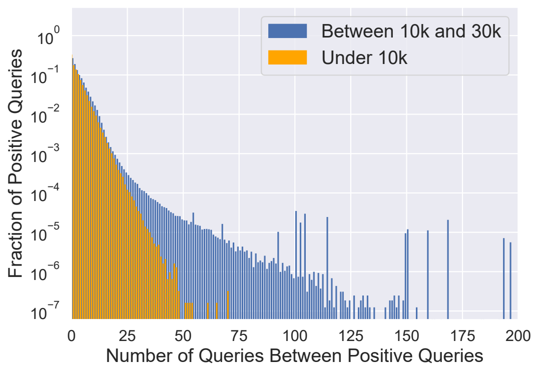

Figure 3: Comparison of the amount of false queries between consecutive positives queries for ciphertexts deciphered in under 10k queries and between 10k and 30k queries (using a perfect oracle

and our settings. In both settings, we want to be able to finish the attack in a limited time period (30 seconds timeout in [64] and 10 hours until ticket expiration in our case). This changes our optimization goals. We don't want to ensure that all attacks succeed (no false positive), but maximize the percentage of attacks that finish successfully below a fixed query number threshold. In Section 5 we show how we can exploit our attack even if we can't finish it in less than 10 hours. However, using the attack acceleration we present in Section 6, an under 10 hours attack becomes practical and minimizing the query overhead becomes significant.

7.1 False Postive Detection

To minimize the query overhead, our novel approach does not try to prevent false positives with costly query retries, but instead we accept a non-negligible false positive rate. If we are able to detect when a false positive occurred, we can instead trace back to a previous correct state of the attack. As we will show, the cost of such a detection and trace back is smaller on average than the cost of reducing the false positive rate.

The main remaining question is how to detect a false positive before the attack finishes. For this, we exploit a statistical property of the attack that is unique to the subset of "fast" messages that can be attacked successfully under our timing constraints before the token expires. We recorded ≈ 40000 RSA ciphertexts generated by the KDC for the AS-REP message. For each ciphertext, we simulated a Bleichenbacher attack with a maximal cutoff threshold of 30000 queries. The attack was able to decrypt ≈ 11500 ($\approx 29\%$) of the ciphertexts in under 30000 queries, and ≈ 3200 ($\approx 8\%$) of the ciphertext in under 10000 queries.

Figure 3 shows the number of false queries between consecutive positive queries. We can see that for the subset of "Under 10k" messages, a sequence of more than 30 consecutive false queries only occurs with a negligible probability, and we don't see any sequences of more than 70 consecutive false queries. Moreover, our experiments showed that if we reach an incorrect state due to a false positive, we will get very long sequences of false queries. This means that we can use such long sequences to detect that a false positive has occurred. Our trace-back algorithm saves the state of the attack for the last 15 positive queries; if a sequence of more than 100 false queries is detected, the algorithm's state is reverted to the earliest saved position. Note that we use conservative numbers to handle multiple false positives.

We developed a low overhead query repetition algorithm tailored to the hit count distribution shown in Table 3. It doesn't require any repetition for the vast majority of both false and negative queries, repeating a query only in a small number of borderline cases. The resulting oracle still has a low but not negligible false positive rate and thus requires our trace-back approach.

Our low overhead repetition algorithm works as follows:

1. A zero hits result is taken as a definite nonconforming query result. It is not repeated. This leads to 7% of false-negative results, but the attack can recover and still works with a slightly larger number of queries.
2. A two hits or more result is taken as a definite conforming query result and is not repeated.
3. A single hit is a borderline result which is repeated up to 3 times:
   (a) If the result of the next query is two hits or more, the repetition stops and the original query is considered conforming.
   (b) If the next query is borderline or 0 hits, we continue to the next query.
4. If no query had two hits or more, the original query is considered nonconforming. This results in a very small number of false-negative results, but again the attack can recover.

Note that our traceback algorithm has one exception. Reverting the first positive query (i.e., the first multiplier found in phase 2 of the attack as described in Appendix A) may require a very large number of additional queries. Thus, for the first positive query, instead of the usual repetition algorithm. As the majority of 5 has only a negligible false positive probability, we never revert it.

7.2 Noisy Oracle Simulation

To test our novel traceback approach, we created a simulated noisy oracle based on the distribution from Table 3. We used this oracle in our simulation to compare our low-overhead traceback approach with two other higher-overhead algorithms. The first is the "double" conservative algorithm that repeats each positive query (hit count > 1) and requires both attempts to be positive to return a positive result. The

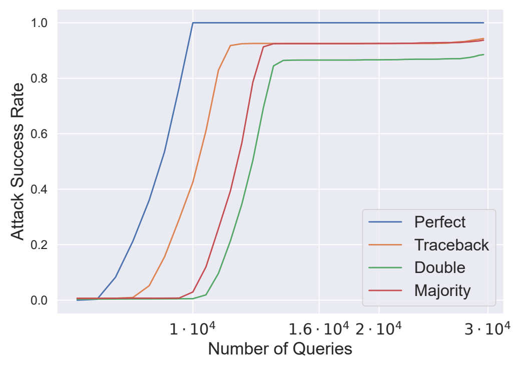

Figure 4: Comparison of noisy oracle handling methods in performed simulation experiment

second is the less conservative "majority" algorithm that, after the first positive query, can attempt up to two more times, requiring a majority of two positives to return a positive result. The "double" algorithm has fewer queries and a lower false positive rate compared to the "majority" at the cost of a higher false negative rate.

Figure 4 shows the results of our simulation comparing the fraction of the attacks that were able to finish successfully with a given number of queries. Note that as we are only interested in "fast" attacks, we limited the simulation to ciphertexts that can be decrypted with less than 10000 queries assuming a *perfect* not noisy oracle, which we also used as our baseline. As we can see, only our traceback-based attack is able to decrypt a non-negligible number of messages in less than 10000 queries with a noisy oracle, reaching a success rate of over 40% of the messages decrypted in less than 10000 queries with a perfect oracle.

7.3 Kerberos Sessions Hijacking

Using our efficient traceback attack to handle the noisy oracle, we were able to exploit the remote vector approach for a full end-to-end attack that allowed us to recover session tokens in less than 6 hours, *before they expire*. The session token can be exploited to gain access to all of the resources and services the user has permission for. For example, if we target an admin user, we can exploit the session token to open a remote powershell with full admin permissions.

We ran our full traceback attack and were able to successfully hijack in less than 10 hours 15 out 17 different session tokens generated by the KDC (taken from the "Under 10k" set). The full experiment results are provided in Appendix B. Using our remote vector, we were able to achieve an average rate of 25.8 queries per minute, which translates to up to $16,000$ queries within a 10-hour span. Our trackback approach combined a low false negative rate of 1.74% achieved a very small query overhead. The overhead for most attacks was under 1500, with several attacks having an overhead of ≈ 1000. This is much lower than the minimal overhead of ≈ 2000 incurred by any approach that requires even a single repetition for a positive query to reduce the false positive rate.

8 Optimizing Attack Retires With Early Abort

To hijack a session, we need to be able to finish the attack and decrypt its token before it expires, i.e., in less than 10 hours. The attack we described in Section 7 is able to perform a little over 16000 queries in 10 hours. This means we can only successfully attack tokens whose messages can be decrypted with less than 16000 *noisy* queries. A naive approach to finding such a message would be simply trying to run the full attack with a 16000 queries (ten-hour) limit on a message from a new session. If the attack doesn't finish in time, we retry the attack with a new message from a new session until the attack succeeds.

Assuming $P_{<16k}$ is the probability that a message can be decrypted with less than 16000 *noisy* queries, such an attack is expected to take on average $\frac{1}{P_{<16k}} \cdot 10$ hours. We will now show how to significantly reduce the overall attack time using our early abort method.

8.1 Early Abort Optimization

Our early abort optimization is based on the following insight: There is a strong correlation between the number of queries required to find the first positive query (i.e., to find the first multiplier in phase 2 of the attack as described in Appendix A) and the probability the message can be decrypted with a small number of queries.

Our optimization runs as follows: We run the attack till we reach a threshold of a fixed number of queries, which we denote Q_{th}. If no positive query has been detected, i.e., the first multiplier was not found, we early abort and rerun the attack on a new message. Otherwise, we continue to run the attack until it either succeeds or we reach the $16k$ query limit. If the attack doesn't succeed, we will again try to attack a new message. The value of Q_{th} has a significant effect on the average complexity of the attack. A lower value will allow us to early abort faster, but will increase the probability we will early abort attacks that could have succeeded. We used a simulation to explore the attack complexity with different values of Q_{th}. Note that setting $Q_{th} = 16k$ results in the naive retry approach.

We tested our attack using a simulated noisy oracle over the ≈ 40000 RSA ciphertexts generated by the KDC, running the attack (without early abort) 5 times for each ciphertext. In each run of the attack, we stored the number of queries required to find the first multiplier, and the number of queries required to finish the attack. Using the results, for each value of Q_{th}, we can calculate the following values:

$P_{<16k}(Q_{th})$ — The probability that, for a random ciphertext,

Q_{th}	$P_{<16k}$	$P_{>16k}$	$Q_{<16k}$	n_{msgs}	T_q
200	0.006	0.0000	8657	160	40578
300	0.009	0.0000	8642	109	41267
400	0.012	0.0001	8619	83	41941
500	0.015	0.0002	8666	66	41352
600	0.018	0.0003	8743	55	41822
700	0.020	0.0003	8763	49	42684
1000	0.029	0.0005	8878	34	42854
2000	0.056	0.0015	9217	17	43080
3000	0.083	0.0027	9528	12	43159
16000	0.178	0.0375	11115	5	84949

Table 5: Comparison of Early Abort Based on the First Multiplier

the first multiplier will be found with less than Q_{th} queries and that the attack will succeed with less than $16k$ noisy queries.

$P_{>16k}(Q_{th})$ — The probability that, for a random ciphertext, the first multiplier will be found with less than Q_{th} queries but the attack will require more than $16k$ noisy queries to succeed.

$Q_{<16k}(Q_{th})$ — The average query count to decrypt a random ciphertext, assuming the first multiplier is found with less than Q_{th} queries and that the attack will succeed with less than $16k$ noisy queries.

$n_{msgs}(Q_{th})$ — The average number of random ciphertext we need till we find a ciphertext such that the first multiplier will be found with less than Q_{th} queries and that the attack will succeed with less than $16k$ noisy queries. calculated as:

$$n_{msgs}(Q_{th}) = \frac{1}{P_{<16k}(Q_{th})}$$

$T_q(Q_{th})$ — The expected number of messages required to finish the full attack. It is calculated as:

$$T_q(Q_{th}) = n_{msgs} \cdot (Q_{<16k} \cdot P_{<16k} + 16000 \cdot P_{>16k} + Q_{th} \cdot (1 - P_{<16k} - P_{>16k}))$$

The resulting values as a function of Q_{th} are shown in Table 5. The last row with $Q_{th} = 16000$ is equivalent to the naive approach that simply tries to attack all messages without early abort. Using low Q_{th} values can reduce the total attack time by half compared to the naive approach. We note that for low Q_{th} values (i.e., below 400), the results are slightly noisy due to the low probability of $P_{>16k}(Q_{th})$. For the experimental verification of the attack, we use a conservative value of 600 for Q_{th}.

8.2 Experimental Results

We implemented our early abort attack and experimentally verified it. In our attack we use a Q_{th} with a value of 600.

#	n_{msgs}	T_q	T[hours]	$Q_{<16k}$
1	36	29336	18.4	8309
2	30	26185	16.4	8680
3	29	27023	16.9	9367

Table 6: Experimental results for token recovery using the early abort attack.

Table 6 shows the number of messages (n_{msgs}), total queries (T_q), and total attack time in hours (T[hours]) for each token recovered. It also shows the number of queries required for the successful decryption of the last message ($Q_{<16k}$). We note that the attack seems to be faster than our simulated attack, with a much lower number of messages required before we are able to recover a token. We believe that this might be due to the distribution of errors. In our simulation, we assume uniform and independent error distribution, but in reality, they may occur in bursts. We leave the analysis of the noise distribution and its effect on such attacks for future work.

9 Conclusions

Modernizing operating systems in general, and their security mechanisms in particular, is a difficult task that requires updating cryptographic implementations while trying to maintain backward compatibility. In Kerberos, this process lead to a convoluted combination of modern and legacy cryptographic primitives and designs. Although the Kerberos protocol is critical to the security of countless sensitive networks, and has undergone countless updates, its default configuration for the most security sensitive use cases (that require 2FA) is still vulnerable to an attack published more than 25 years ago.

9.1 The Danger of Legacy Crypto

Our research can highlight the many different ways legacy crypto can be exploited to implement a full end-to-end attak. We will now discuss some of the root causes we identified and how we propose to mitigate them.

Using Nonconstant-time Crypto Legacy Crypto such as the RSA PKCS #1 v1.5 based encryption is notoriously hard to implement securely. In more than 25 years we have seen countless cycles of new attacks and new mitigations (especially but not limited to TLS). While it is possible to use the lessons learnt in TLS and try and implement the decryption with constant-time code, and add the required Bleichenbacher mitigations (as was done in the MIT implementation), we believe this is not the right path to move forward. A better alternative such as ECDSA and ECDH based options are already supported in Kerberos, we should push for complete deprecation of RSA PKCS #1 v1.5.

Unauthenticated Data In our attack we exploited the fact that many parts of Kerberos messages are not authenticated (e.g., to spoof both AS-REP and error messages). This was also exploited in the downgrade attack by Google Project Zero [22]. Although a complete overhaul of Kerberos to include strong authentication might be a challenging task, there are other options to reduce the risk. For example, Kerberos can be used over more modern and vetted protocols such as TLS [32]. Using such options can help prevent attackers from exploiting such vulnerabilities in the Kerberos protocol.

Lack of Forward Secrecy Protocols such as SMB3 rely on Kerberos for authentication and key distribution. To protect the sensitive data of such protocols, Kerberos should ensure forward secrecy. This can be done with a redesign that adds forward secrecy to the protocol, such as incorporating ephemeral Diffie-Hellman key exchange or tunneling the communication using a modern and secure protocol such as TLS (similar to the proposal for Kerberos above). In any case, the security properties of the protocols that rely on Kerberos should be well-defined and communicated to the users.

Unlimited Access to 2FA token Although it is not a weakness by itself, our attack relies on the fact that there is no global rate limit to the number of calls to the smartcard. Although an initial PIN code entry is required, we were able to cause the client machine to use the smartcard to decrypt more than 10000 RSA ciphertexts without any user interaction. This is several orders of magnitude more than what we can expect in a normal operation. Smartcards should provide an option to configure some policy for rate limitations. Moreover, the OS can also force user interaction by limiting the amount of calls to the smartcard before asking the user to reenter their PIN code, this should be implemented using a global counter.

9.2 Further Research

In this work, we provide an efficient algorithm to lower the Bleichenbacher's attack overhead when using noisy oracles. Our algorithm is focused on efficiency in the subset of "fast" attacks that require a relativity low number of queries. This is motivated by time constrained attack scenarios where slower attacks will fail to finish on time in any case. As we have shown, other approaches might improve the success rate when we consider other subsets of messages. We believe that further exploration of the problem space will reveal many more interesting approaches and tradeoffs that can optimize the attack for more real world scenarios.

Acknowledgments

We thank Orr Dunkelman and Yossi Oren for the helpful discussions and Eitan Lezmy for his technical help with testing various versions of Windows. This project has been supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik ICRC, Robert Bosch Technologies Israel Ltd, and ISF grant no. 1807/23.

References

[1] CVE-2022-4182., November 29 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-4182.

[2] CVE-2023-1814., April 4 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-1814.

[3] CVE-2023-1823., April 4 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-1823.

[4] CVE-2023-2459., May 2 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-2459.

[5] CVE-2023-2460., May 2 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-2460.

[6] CVE-2023-2940., May 30 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-2940.

[7] ISO/IEC JTC 1/SC 17. Identification cards — Integrated circuit cards — Part 15: Cryptographic information application. Standard, International Organization for Standardization, Geneva, CH, June 2016.

[8] Apple. Applosxkrb5, 2023. https://opensource.apple.com/source/Kerberos/Kerberos-62/KerberosClients/KerberosApp/Documentation/using-osx.html?f=text.

[9] ArchLinux. Arch krb5, 2023. https://wiki.archlinux.org/title/Kerberos.

[10] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium*, August 2016.

[11] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. Research Report RR-7944, INRIA, April 2012.

[12] Tripwire Blake Strom, Microsoft 365 Defender; Travis Smith. Pass the hash - blake strom, microsoft 365 defender; travis smith, tripwire, 2020. https://attack.mitre.org/techniques/T1550/002/.

[13] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *CRYPTO*, 1998.

[14] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of Bleichenbacher's oracle threat (ROBOT). In *USENIX Security Symposium*, 2018.

[15] Livia Capol. Experimenting with the bleichenbacher attack. 2021.

[16] Cern. Cern centos, 2023. http://lxsoft102.cern.ch/cern/centos/7.3/cr/x86_64/repoview/system_environment.libraries.group.html.

[17] CSO. Cso-the 15 biggest data breaches of the 21st century, 2022. https://www.csoonline.com/article/534628/the-biggest-data-breaches-of-the-21st-century.html.

[18] Omkar Dastane, Kinn Bakon, and Zainudin Johari. The effect of bad password habits on personal data breach. *International Journal of Emerging Trends in Engineering Research*, 8:6950–6960, 10 2020.

[19] SpiderLabs; Edward Millington Ed Williams, Trustwave. Os credential dumping: Lsass memory, 2020. https://attack.mitre.org/techniques/T1003/001/.

[20] Enlyft. Companies using active directory, 2023. https://discovery.hgdata.com/product/microsoft-active-directory.

[21] Fedora. Fedora krb5, 2023. https://fedoraproject.org/wiki/Kerberos_KDC_Quickstart_Guide.

[22] James Forshaw. rc4 is still considered harmful, 2022. https://googleprojectzero.blogspot.com/2022/10/rc4-is-still-considered-harmful.html.

[23] FreeBSD. Freebsd heim, 2023. https://docs.freebsd.org/en/books/handbook/security/#kerberos5.

[24] Gentoo. Gentoo krb5, 2023. https://wiki.gentoo.org/wiki/Kerberos_Windows_Interoperability.

[25] HGInsights. Hgdata companies using active directory, 2023. https://discovery.hgdata.com/product/microsoft-active-directory.

[26] IBM. Ibm krb5, 2023. https://www.ibm.com/docs/en/spectrum-conductor/2.4.1?topic=setup-installing-kerberos-server-client.

[27] Naomaru Itoi and Peter Honeyman. Smartcard integration with kerberos v5. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, WOST'99, page 7, USA, 1999. USENIX Association.

[28] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In *ESORICS*, 2012.

[29] Assar Westerlund Johan Danielsson. Heimdal kerberos implementation, 1998. https://www.heimdal.software/.

[30] Assar Westerlund Johan Danielsson. Heimdal—an independent implementation of kerberos 5, 1998. https://www.usenix.org/legacy/publications/library/proceedings/usenix98/freenix/heimdal2.pdf.

[31] Jakob Jonsson and Burt Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, February 2003.

[32] Simon Josefsson. Using Kerberos Version 5 over the Transport Layer Security (TLS) Protocol. RFC 6251, May 2011.

[33] Hubert Kario. Everlasting ROBOT: the marvin attack. *ESORICS*, page 1442, 2023.

[34] Sowmya Karunakaran, Kurt Thomas, Elie Bursztein, and Oxana Comanescu. Data breaches: User comprehension, expectations, and concerns with handling exposed data. In *SOUPS @ USENIX Security Symposium*, 2018.

[35] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In *USENIX Security Symposium*, 2023.

[36] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking rsa-based sessions in SSL/TLS. In *CHES*, 2003.

[37] Yeu-Pong Lai and Wei-Feng Wu. The defense in-depth approach to the protection for browsing users against drive-by cache attacks. *Secur. Commun. Networks*, 8(7):1422–1430, 2015.

[38] Wenting Li, Ping Wang, and Kaitai Liang. Hpake: Honey password-authenticated key exchange for fast and safer online authentication. *Trans. Info. For. Sec.*, 18:1596–1609, jan 2023.

[39] Tim Medin. Kerberoasting attack - tim medin, 2015. https://github.com/nidem/kerberoast/.

[40] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *USENIX Sec*, 2014.

[41] Microsoft. server message block (smb) protocol - microsoft, 2021. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SMB/[MS-SMB].pdf.

[42] Microsoft. Active directory technical specification - microsoft, 2022. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-ADTS/[MS-ADTS].pdf.

[43] Microsoft. Kerberos protocol extensions - microsoft, 2022. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-kile/2a32282e-dd48-4ad9-a542-609804b02cc9.

[44] Microsoft. server message block (smb) protocol version 2 and 3 - microsoft, 2022. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SMB2/[MS-SMB2].pdf.

[45] Microsoft. Ticket lifetimes - microsoft, 2022. https://learn.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/maximum-lifetime-for-user-ticket?source=recommendations.

[46] Microsoft. whats-new-in-credential-protection, 2022. https://learn.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/whats-new-in-credential-protection.

[47] MIT. Mit kerberos implementation - mit, 2002. https://web.mit.edu/kerberos/.

[48] MIT. Mitheim, 2007. http://web.mit.edu/macdev/KfM/Common/Documentation/download.html.

[49] MIT. dictionary attacks and mitigations, 2023. https://web.mit.edu/kerberos/krb5-latest/doc/admin/dictionary.html.

[50] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[51] Collins W. Munyendo, Yasemin Acar, and Adam J. Aviv. "in eighty percent of the cases, i select the password for them": Security and privacy challenges, advice, and opportunities at cybercafes in kenya. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[52] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, dec 1978.

[53] Dr. Clifford Neuman, Sam Hartman, Kenneth Raeburn, and Taylor Yu. The Kerberos Network Authentication Service (V5). RFC 4120, July 2005.

[54] Dr. Clifford Neuman and Theodore Ts'o. The Kerberos Network Authentication Service (V5). RFC 1510, September 1993.

[55] OpenSuse. Opensuse krb5, 2023. https://doc.opensuse.org/documentation/leap/security/html/book-security/cha-security-kerberos.html#sec-security-kerberos-admin-kdc.

[56] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *CCS*, pages 1406–1418. ACM, 2015.

[57] PCWorld. Windows 10 continues to dominate windows 11, 2023. https://www.pcworld.com/article/2094755/windows-10-doesnt-give-windows-11-a-chance.html.

[58] Tim Polk and Sean Turner. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176, March 2011.

[59] Andrei Popov. Prohibiting RC4 Cipher Suites. RFC 7465, February 2015.

[60] RedHat. Redhat krb5, 2023. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system-level_authentication_guide/configuring_a_kerberos_5_server.

[61] RedHat. Redhat krb5, 2023. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system-level_authentication_guide/krb-smart-cards.

[62] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[63] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[64] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of bleichenbacher's CAT: new cache attacks on TLS implementations. In *IEEE Symposium on Security and Privacy*, 2019.

[65] Samba. Sambaheim, 2022. https://www.samba.org/samba/security/CVE-2022-3437.html.

[66] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Financial Cryptography*, 2017.

[67] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*, 2019.

[68] Franco Tommasi, Christian Catalano, and Ivan Taurino. Browser-in-the-middle (bitm) attack. *Int. J. Inf. Sec.*, 21(2):179–189, 2022.

[69] Ryan Becwar; Vincent Le Toux. Pass the ticket, 2020. https://attack.mitre.org/techniques/T1550/003/.

[70] Brian Tung and Larry Zhu. Public Key Cryptography for Initial Authentication in Kerberos (PKINIT). RFC 4556, June 2006.

[71] Ubuntu. Ubuntu krb5, 2023. https://ubuntu.com/server/docs/kerberos-introduction.

[72] Working Group WG1363. Ieee standard specifications for public-key cryptography. *IEEE Std 1363-2000*, pages 1–228, 2000.

[73] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *CCS*, 2017.

[74] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *USENIX Security*, 2014.

A Bleichenbacher Attack algorithm

Below is a description of how an attacker can use the Bleichenbacher oracle Bl to perform an RSA secret key operation, such as decryption, on c without knowing the secret exponent d. We refer the reader to [13] for a more complete description.

High Level Attack Description Let c be an integer. To compute $m = c^d \bmod N$, the attack proceeds as follows:

Phase 1: Blinding The attacker repeatedly iterates over integers beginning at s_0 and computes $c^* \leftarrow c \cdot s_0^e \bmod N$. The attacker checks if c^* is a conforming ciphertext by evaluating $\mathsf{Bl}(c^*)$. When an s_0 is found such that $\mathsf{Bl}(c^*) = 1$. The phase ends. This step can be skipped completely if c is already a valid PKCS #1 v1.5 ciphertext in which case $s_0 = 1$.

We note that when the oracle returns a positive result ($\mathsf{Bl}(c^*) = 1$) the attacker knows that the corresponding message $m^* = m \cdot s_0 \bmod N$ starts with 0x0002. Thus, it holds that $m \cdot s_0 \bmod N \in [2B, 3B)$ where $B = 2^{8(\ell-2)}$ and ℓ is the length of N in bytes. Finally, the condition of $m \cdot s_0 \bmod N \in [2B, 3B)$ implies that there exists an integer r such that $2B \leq m \cdot s_0 - rN < 3B$, or equivalently:

$$\frac{2B + rN}{s_0} \leq m < \frac{3B + rN}{s_0}.$$

Phase 2: Range Reduction Having established that $\frac{2B+rN}{s_0} \leq m < \frac{3B+rn}{s_0}$, the attacker proceeds to choose a new integer multiplier s, computes $c^* \leftarrow c \cdot s^e \bmod N$ and checks that $\mathsf{Bl}(c^*) = 1$. When a suitable s is found, the adversary can further reduce the possible ranges of m, further detailed in [13]. The attack terminates when the possible range of m is reduced to a single candidate.

The original algorithm was published in 1998 and was known as the 'Million Message Attack'. However, since then there have been many optimizations for the attack [11]. In some implementations, not all parts of the padding scheme are verified and the attack can be adapted.

B Session Hijacking Experimental Results

Table 7 shows the experimental results of our full end-to-end session hijacking attack. For each successful attack, we give the attack time, the total queries required for the attack (including repetitions as described in Section 4.1). We also provide a baseline of the number of required queries assuming a perfect oracle. Two attacks were stopped due to errors that prevented them from finishing on time.

Table 7: Experimental Attack Results

Session	Success	Attack Time (hours)	Total Query	Perfect Oracle Query
1	True	5.44	8422	7203
2	True	4.89	7805	7362
3	True	5.96	9151	7587
4	True	5.64	8787	7713
5	True	5.22	8180	7178
6	True	5.88	8956	7797
7	True	5.71	8746	7291
8	False	-	-	8363
9	True	5.52	8512	7310
10	True	5.63	8742	7340
11	True	6.13	8864	7245
12	False	-	-	7192
13	True	5.09	7806	7230
14	True	5.48	8603	7645
15	True	5.66	8992	7919
16	True	6.30	9889	9070
17	True	7.43	11449	9449