MEASURING THE EFFECTIVENESS OF MICROSOFT AUTHENTICODE: A SYSTEMATIC ANALYSIS OF SIGNED FREEWARE

Mina Jafari

A thesis
in
The Department
of
Concordia Institute of Information Systems and Security

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
Concordia University
Montréal, Québec, Canada

September 2020

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mina Jafari**

Entitled: **Measuring the Effectiveness of Microsoft Authenticode: A Systematic Analysis of Signed Freeware**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Amr Youssef

_____ Examiner
Dr. Amr Youssef

_____ External Examiner
Dr. Otmane Ait Mohamed

_____ Supervisor
Dr. Mohammad Mannan

Approved by _____
Dr. Mohammad Mannan, Graduate Program Director

September 3, 2020 _____
Dr. Mourad Debbabi, Interim Dean
Gina Cody School of Engineering and Computer Science

Abstract

Measuring the Effectiveness of Microsoft Authenticode: A Systematic Analysis of Signed Freeware

Mina Jafari

Recent studies have shown that Authenticode, the Windows code signing standard for portable executable files, can be abused by potentially unwanted programs (PUP) and malware to evade detection and bypass Windows protections. These studies discuss improper signature checks by frameworks (e.g., anti-virus programs), key mismanagement, improper verification by certificate authorities (CAs) and underground certificate trade as weaknesses that can be abused in Windows code signing public key infrastructure (PKI). We explore the Authenticode signatures of supposedly benign applications in the wild to gain a clearer understanding of this mechanism so that we can identify potential issues that can undermine trust in Authenticode. For studying the blackbox of the Authenticode, we tackle the main challenge of doing a measurement study on Authenticode, lack of a comprehensive corpus of Windows code signing certificates. As placing trust in the freeware that is downloaded from web is one significant use case of code signing, we target eight popular download portals as source of our dataset and collect 106K Windows applications. We present an analysis framework for studying code signing certificates and extract 27K certificates from signed executable applications. This framework provides a crawler for automated download of applications from download portals. Furthermore, as part of our analysis framework, we develop a linter that is specifically designed for Authenticode certificates. Both of our tools are in the process of release for public use of researchers. Our results identify issues in the code signing certificates that the Authenticode validation fails in preventing them. Usage of inadequately secure hash and public key algorithms such as MD5, SHA1 and 1024-bit RSA, missing or invalid Key Usage and Extended Key Usage, missing revocation information, non-critical Basic Constraints for CA certificates are examples of the issues that potentially undermine both integrity and authenticity assurance that Authenticode provides.

Acknowledgments

I would like to sincerely thank my supervisor Dr. Mohammad Mannan for his guidance and availability throughout this study.
I am also grateful to my beloved parents, my caring, awesome sisters and my supportive friends, for their constant support, kindness and insights they provided me throughout my life.
I would like to dedicate this thesis to my parents for their endless love, support and encouragement in my moments of crisis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Overview

When we download third-party software from web, how can we decide to trust the application that will be installed and/or executed on our computer? Code signing PKI can assure us about authenticity and integrity of an application. Windows code signing verifies if an application has been signed by a legitimate author and has not been tampered after applying the signature. If Authenticode signature is verified successfully, name of the authenticated publisher will be provided to the user. Then it is up to the user's decision if he wants to install or launch an application from the authenticated publisher. If Authenticode signature is not verified, the user will be warned that the application comes from an unknown publisher.

Signed applications can bypass system protections such as Microsoft Defender SmartScreen [33] as well as browser protections and anti-virus programs that forgo scanning of signed binaries. Signed malware such as Stuxnet [11] and Flame [32] or the recent destructive wiper ZeroCleare [19] that leverages a signed driver to bypass Windows hardware abstraction layer are examples of established trust and privilege that is given to the signed binaries by system protections.

The trust that code signing establishes mainly relies on PKI, the same infrastructure used by SSL/TLS certificates. Numerous studies and testing approaches [3, 5, 7, 10, 17, 36] that were conducted on SSL/TLS certificates have revealed cases of certificate misissuance as well as vulnerabilities in different implementations of certificate validation. However, code signing certificates have not been tested in a

systematic way against similar vulnerabilities that can potentially undermine the effectiveness of Windows code signing. A comprehensive set of Code signing certificates cannot be easily collected. In contrast, TLS certificates can be collected simply by scanning the IPv4 address space of search engines such as Censys [37]. Moreover, there is no systematic logging service such as Certificate Transparency (CT) for Windows code signing certificates. On the other hand, Windows code signing has one single propriety implementation unlike SSL/TLS that have several implementations and libraries that are being used by numerous open source products. In consequence, SSL/TLS implementations have been exercised and tested more compared to the closed source implementation of the Authenticode.

Prior studies on Windows code signing relied on the datasets of certificates that were extracted from malicious or potentially unwanted binaries. To achieve a more generic view of the issued certificates in the wild, we target supposedly benign applications as the source of our dataset. We collect a dataset of 106,623 applications that are distributed on the web. Filtering the unique signed applications that contain executable files results in obtaining 27,789 unique signed applications. We analyze Authenticode signatures of these applications as a subset of issued signatures in the wild. We examine Authenticode signatures of these applications to find discrepancies and violations that can breach security of Windows code signing. More importantly we want to shed light on the flaws that are not prevented by Windows Authenticode validation.

As mentioned earlier, the lack of a comprehensive dataset of code signing certificates and the closed source implementation of the Authenticode validation make it more challenging to evaluate Windows code signing. Besides, available documentation for the Wintrust library and related APIs are also limited. Thus, they cannot provide a detailed view of Authenticode validation's logic. On the other hand, not all the error messages are descriptive or clear enough to certainly identify the source of the problems. All these issues would add up to the unclear understanding and limited visibility to Windows code signing. So, for acquiring a clearer insight, we conduct an external evaluation on Windows Authenticode. We take an approach similar to the adversarial testing of the SSL/TLS implementations done by Brubaker et al. [5]. They provided artificial test certificates to several SSL/TLS implementations and libraries and used the differences of the test results as oracles for identifying

flaws in the implementation. Differential testing is not applicable to evaluation of the Authenticode since it is the only propriety implementation of Windows code signing from Microsoft. However, we leverage the idea of generating synthesized test certificates. This enables us to also test Authenticode against unconventional certificates that are not seen in the wild. This method also alleviate the lack of a comprehensive dataset. Authenticode signatures are supposed to be in compliance with the Baseline Requirements for the Issuance and Management of Code Signing Certificates [13]. So, we develop a set of test cases according to these requirements and examine our corpus of code signing certificates as well as our synthesized certificates against these tests. We determine the violations according to the failed tests and determine if the Authenticode validation would prevent them.

Our tests show breaches of trust in both of the authenticity and integrity assurance that Authenticode provides. In terms of integrity, the Authenticode validation still accepts the broken MD5 and SHA1 hash algorithms. An adversary can exploit MD5 collision attack to use a forged certificate for signing malicious code. In other words, an adversary does not need to invest more effort and resources for finding an exploit for SHA1 hash algorithm, which is also accepted by Authenticode. Padded Authenticode signatures is also a known vulnerability which is still seen in the wild. This vulnerability can be abused for breaking the integrity of software without altering the signature. An Authenticode signature itself is not involved in the calculation of the binary's digest. Moreover, the Authenticode signature is encapsulated in a specific data structure padded with zeros, so that the length of the data structure becomes multiple of eight bytes. This design allows an adversary to modify a binary without invalidating the Authenticode signature. We also observe usage of inadequately secure public key algorithms suh as 512-bit and 1024-bit RSA in our corpus of data while the Authenticode validation accepts them. In terms of authenticity, we observe several violations regarding the certificate extensions and constraints. For example, a certificate that may not be authorized for code signing (missing both Key Usage and Extended Key Usage extensions) can validate successfully. Unspecified certificate policy, violated basic constraints, non-critical Key Usage, invalid bit settings for the Key Usage and unavailable revocation information are some other examples of these violations. Moreover, the Authenticode validation does not maintain checks to prevent these violations. Table 1 summarizes our findings and their corresponding

Component	Finding	Implication
X.509 Certificate Version	Certificates can be of version one	There is no way to check if a certificates is authorized for code signing
Certificate Policy Extension	This extension is missing in the leaf certificates	There is no way to check if policy of the certificate is valid
Basic Constraints Extension	Certificates contain two Basic Constraints; one with the CA bit set to true and the other one with the CA bit set to false	An end-entity certificate can take action as a rogue CA
	CA certificates have non-critical Basic Constraints extension	Authorization check can be disregarded if this extension has an unrecognizable value
	Certificates without any Basic Constraints, Key Usage and Extended Key Usage extensions can validate	Authorization check can be bypassed
Key Usage Extension	Leaf certificates do not have this extension	Authorization check can be bypassed
	Leaf certificates have non-critical Key Usage Extension	Authorization check can be disregarded, if this extension has an unrecognizable value
	Leaf certificates violate the required bit settings for the Key Usage extension	An unauthorized certificate such as a timestamping certificate can be used for code signing
Extended Key Usage Extension	Leaf certificates have invalid value set for this extension	An unauthorized certificate such as a TLS server authentication can be used for code signing
Revocation Distribution Points	Leaf certificates do not contain any OCSP or CRL points	Revocation information will not be available for such certificates. Thus, they can remain valid despite of being revoked
	OCSP and CRL points have unreachability issues	Revocation status will not be taken into account for the Authenticode validation. Thus, it is likely that a revoked certificate would be deemed valid

Table 1: Summary of Findings

implications that can undermine effectiveness of code signing.

It is noteworthy that our findings can be classified into two folds: findings that are observed in real valid certificates and findings that are observed in our valid synthesized certificates. For example, our finding regarding the certificates of version one has not been observed in the wild, however, Authenticode validation accepts a code signing certificate of version one. Thus, since it can be abused by an adversary, we report it as a security issue. On the other hand, identifying the violations relies on the effective date of the constraints and the issue date of the certificates. So, since Table 1 intends to provide a summarized view of our findings, we mention each finding as a generic statement. The quantitative numbers of the occurrences are presented in Chapter 4.

1.2 Thesis Focus

We intend to examine a subset of Authenticode certificates that are used for signing supposedly benign applications. Authenticode validation code is propriety and since detailed steps of the validation are not clear, there is a need for an external evaluation of this mechanism. Besides, X.509 system, the underlying component of Authenticode, is prone to vulnerabilities which has been uncovered in studies related to SSL/TLS certificates. We aim to determine if this mechanism is functioning properly and consistently. For this end, we investigate code signing certificates for potential issues and examine Authenticode validation functionality while exposed to any potential vulnerabilities. We use our collected corpus of data to study specification violations seen in the wild. We report potential violations as evidence of CAs' misissuance or bad practices of software publishers. Furthermore, if any potential violation is validated successfully, we specify shortcoming of Authenticode validation in preventing security critical issues. The primary objective of this thesis is to evaluate the effectiveness of Windows code signing mechanism by systematical analysis of a corpus of signed applications distributed in the wild. As part of this objective, we aim to answer the following research questions:

Question 1. How prevalent is misissuance of code signing certificate in the wild or in other words, how conformant are issued Authenticode signatures to the code

signing baseline requirements?

Question 2. Does Authenticode validation prevent potential violations and flaws?

Question 3. What are design features of Authenticode that can be abused by an adversary?

1.3 Contributions

1. We tackle two main challenges of doing a measurement study on the Windows code signing mechanism: the lack of a comprehensive dataset, and the limited visibility to the Authenticode's closed source implementation. We design a framework for collecting signed applications distributed by third party software publishers on the web. We design an analysis framework which leads to the development of a certificate linter specifically for Authenticode. We are in the progress of publicizing both of our frameworks for researchers.

2. We conduct an exploratory analysis on the effectiveness of Windows code signing in the wild.

   (a) Our results show existence of violating certificates in the wild. These violations are involved with critical extensions and constraints of code signing certificates such as Basic Constraints, Key Usage, Extended Key Usage and revocation distribution points. More importantly, we highlight that the Authenticode validation does not prevent violating certificates in the wild. In other words, we shed light on the lack of effectiveness of Authenticode regarding both of the authenticity and integrity assurance that it is supposed to maintain for the end-user. Usage of a weak hash algorithm such as MD5 and SHA1 is still observed in valid code signing certificates in the wild, for example, 94 applications in our dataset used MD5 as digest algorithm as late as 2017. Weak public key algorithm such as 1024-bit RSA is not observed in the valid certificates of our dataset, however, our tests shows that it is accepted by Authenticode validation.

   (b) We report evidence consistent with the code signing certificates misissuance of certificate authorities. Analysis of 27,789 unique signed applications reveals violations in requirements and constraints of code signing

certificates that can potentially undermine the effectiveness of the Authenticode. Our results shows that these violating certificates were issued as recent as 2017, 2018, and 2019 by known authorities such as GlobalSign, Comodo, Symantec, Digicert, Microsoft, Intel and Dell.

1.4 Outline

Next chapters of this thesis are organized as follows. Chapter 2 provides necessary background and literature related to this dissertation. In Chapter 3, we describe our approach for studying effectiveness of Authenticode in the wild. We introduce our data collection and analysis frameworks and tools. Chapter 4 presents statistics regarding our dataset as well as results of our tests and analysis. Eventually, Chapter 5 concludes.

Chapter 2

Background and Related Work

2.1 Background

In this section, we present the necessary background information for the Authenticode and its known potential abuse.

2.1.1 Code Signing

Code signing is a specific use of certificate-based digital signatures that enables end-users to verify the identity of the software publisher and confirm that the software has not been modified since it was signed. Operating systems (OS) such as Microsoft Windows, MacOS and Android validate digital signatures during installation or execution of programs that request to run with high privileges. Warnings about unsigned code can cause end-users to abandon installation or execution. MacOS manages code signing by Xcode[1] and Codesign. Android also has three schemes for APK signing.[2] Code signing is critical for IoT (Internet of Things) devices as well. The firmware, operating system and application update processes are highly sensitive and can be a major target for attackers. Companies provide different code signing solutions and services for IoT.

We intend to study the code signing standard of Microsoft Windows; since Windows is the most commonly installed desktop OS and has the largest number of users

[1] `https://developer.apple.com/documentation/xcode`
[2] `https://source.android.com/security/apksigning`

globally. Code signing can significantly aid to protect many users against cyber security attacks that involve malicious applications. Likewise, any vulnerability in the code signing mechanism can impact a large number of systems.

2.1.2 Authenticode Overview

Microsoft Authenticode is the Windows code signing standard for portable executables (PE). When an application carries a valid digital signature, it means that both of the identity of the author and the integrity of the software are verified. However, the signature itself cannot verify intent of the software. Ensuring the end-user about the identity of the software publisher is one of the use cases of digital signature. Furthermore, this signature is used as an indicator of clean reputation by system and web protections as well as anti-virus programs. In other words, these protection mechanisms treat unsigned files with more suspicion.

Authenticode is based on the public-key cryptography standard (PKCS)#7 and uses X.509 v3 certificates [14] to bind identity of the software publisher to the signed file. The established trust is anchored in the assurance of the X.509 v3 certificate that is issued by a publicly trusted certificate authority. Furthermore, signed code is encouraged to be countersigned by a timestamping authority. A timestamp maintains lifetime validity for the signed code. In other words, the expiration of the code signing certificate will not cause the Authenticode signature to expire. However, if a certificate authority revokes a certificate for reasons such as private key compromise, this certificate will be no longer valid regardless of its timestamp or validity period. These are components of Authenticode infrastructure. Any breaches in these components can undermine effectiveness of Authenticode mechanism.

2.1.3 Authenticode Structure

Authenticode signature, which is comprised of the hash value of the portable executable (PE) file, a signature generated by the software publisher's private key and the X.509 certificates of the publisher is contained in the PKCS #7 SignedData structure and is appended to the end of the PE file. Furthermore, a description and/or the URL of software publisher and a timestamp may be added to the signature optionally. This PKCS #7 data is encapsulated in WIN_CERTIFICATE structure (declared in

Wintrust hedear file). WIN_CERTIFICATE structure is padded by zeros so that its length be a multiple of eight. Thus, its length is not always same as length of the PKCS #7 data that contains signature. For calculating hash of a PE file, three fields of the PE file are skipped: The PE file's checksum, the Authenticode signature itself and the pointer to Authenticode signature (the red fields indicated in Figure 1). Hence, modification of these specific fields will not alter the signature.

2.1.4 Authenticode Validation

WinVerifyTrust [16] API can be used to verify signature of a PE file. For this purpose, the trust provider of this function is set to the WINTRUST_ACTION_GENERIC_VERIFY_V2 policy which specifies the criteria that needs to be satisfied for Authenticode verification. We explain how signatures are verified against this policy according to Windows documentation [14]. Integrity of PKCS #7 that is containing the signature will be verified according to PKCS #7 Cryptographic Message Syntax standard. Then certificate of the software publisher is required to be verified. Thus, the certificate chain is built to a trusted root certificate using X.509 chain building rules.[3] The signing certificate must contain code signing value for extended key usage (EKU) or the entire chain must contain no EKUs. The certificate must be within its validity period or it should carry a valid timestamp. Furthermore, revocation checking is optional but it is used in many Windows applications and components such as Signtool. The final step is comparing the original hash value of the PE file with the signed hash. If these two hash values do not match, it implies that the file has been altered after it was signed and signature validation will fail.

2.1.5 Known Potential Abuse

As mentioned before when Authenticode skips some places of the executable file (the PE file's checksum, the Authenticode signature itself, and the pointer to the signature) for calculating hash of the PE file. The reason is that these places need to be modified after the file is signed. Thus, modification of these specific fields will not invalidate the signature. As padding signature with extra data, post-signing would not change the signed code, if for any reason part of a program's source code refers to this location, it

[3]Specified by IETF RFC 3280

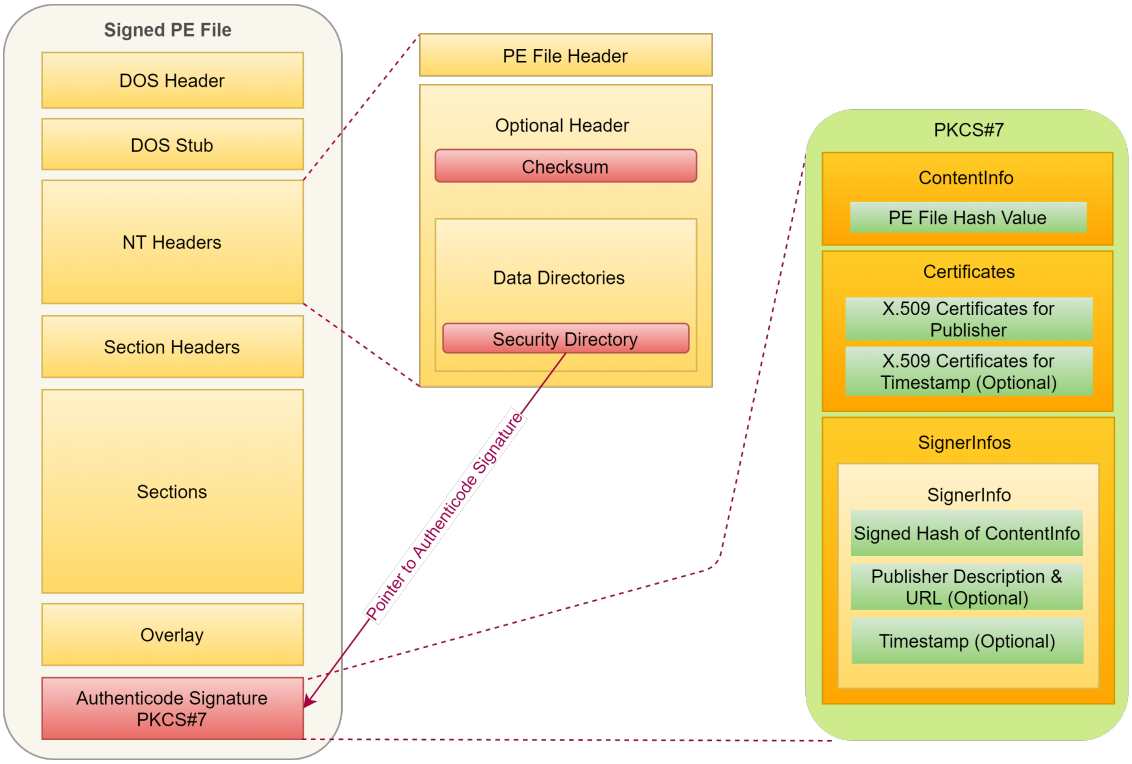

Figure 1: Overview of Signed PE File Format [14].

can be exploited by an adversary. Reports were submitted to Microsoft that software has used this space to embed URLs for downloading installers. These embedded URLs were abused by adversaries for downloading malware. We examine our corpus of Authenticode signatures to see how prevalent are padded signatures in the wild. This vulnerability is intended to be caught by Microsoft as malformed signature error (error code 0x80096011).

2.2 Related Work

We discuss related work in two key areas: measurement studies on Windows Code signing PKI and measurements on TLS certificates.

2.2.1 Windows Code Signing PKI

Sophos [38], Kotzias et al. [24], Alrawi et al. [1] and Kim et al. [22] studied code signing PKI abuse by PUP and malware. They showed that suspicious files were

signed using legitimate certificates issued by trusted CAs.

Kotzias et al. [24] performed a static analysis on Authenticode abuse and effectiveness of existing defenses. They analyzed 356K samples collected from malware datasets (mainly from VirusShare [30]) between 2012 to 2015. 96% of their samples have been flagged as malicious or potentially unwanted by more than three engines on VirusTotal [15]. They studied the digital signature, certificate chain, certificate revocation and timestamp of signed binaries after filtering out benign samples. They used these static features to classify and cluster their samples. They also gathered a list of abused certificates that are still valid. Furthermore, they highlighted a problem with the Authenticode that allowed a timestamped signed binary validate successfully even after its code signing certificate has been revoked. They proposed hard revocation as a solution.

Similar to the work of Kotzias et al. [24], Alrawi et al. [1] explored attributes of certificates for the aim of classification and characterizing malware. Their dataset included over 3 million malware collected from a commercial feed during one month (July 2015). They provided a high-level overview of selected attributes obtained from processing of malicious samples. These attributes included issue and expiry dates, validation duration, chain length, CA counts, TSA counts, issuing CAs, country, and common name. Their analysis intended to provide a representation of signed malware population by calculating number of occurrences of incidents in their corpus of certificates.

Kotzias et al. [24] and Alrawi et al. [1] both reported that majority of the signed binaries in their datasets were PUPs and they concluded that malware was not prevalently signed. Kim et al. [22] did a measurement study on the code signing abuse; specifically on the commonly used methods and the security consequences of the abuse. They categorized root causes of Authenticode abuse by malware to three classes of weaknesses in code signing PKI: inadequate client-side protections, publisher-side key mismanagement and CA-side verification failures. Another study by Kozák et al. [25] particularly highlighted underground trade as another method that allows malware to acquire a valid code signing certificate.

Kim et al. [2] did a measurement study on effectiveness of revocation, as a primary mitigation method against Authenticode abuse. They collected a dataset from three sources publicly released by prior studies [24, 1, 9] and Symantec internal repository

which is proprietary. Thus, the collected data is mostly extracted from abusive certificates. They collected the largest dataset of code signing certificates and they did the first measurement study on the revocation process of Authenticode. The result of their analysis was findings about the three involved roles in the revocation process: Tracking down the abusive certificates, effective revocation of these certificates, and publicizing the revocation information. Significant delay in discovery of compromised certificates and updating revocation information, wrong revocation dates, unreachability and dysfunction of CRL, and OCSP servers were instances of their findings. All these shortcomings would lead to successful validation of a signed malware which undermines security and effectiveness of Authenticode.

Compared to prior studies that mainly focus on suspicious binaries, we examine code signing certificates of supposedly benign applications. For collecting benign applications, we are inspired by prior work [31]. Rivera et al. [31] assessed abuse in distributed software by download portals, websites that categorize and host free proprietary software for download. Their goal was understanding the percentage of undesirable programs distributed by download portals. They reported percentage of PUP and malware in 20 download portals. They quantified number of binaries that were detected by at least one engine on VirusTotal [15] as well as number of binaries that were detected by more than three engines, so that they Could provide a more conservative ratio. They reported a ratio for the observed PUP and malware across all the studied portals ranging between 8% (conservative) and 26% (lax). After collecting the binaries, they processed and executed these files in a sandbox. The processing included extracting static information about the file such as the name and type of the file, collecting the scan report of the file from VirusTotal, decompressing the file's archive and checking of Authenticode signature for signed binaries. They processed Authenticode signature of signed applications at the final step of their analysis for confidently identifying publisher of software. They used Microsoft-provided Authenticode validation tool for verifying the Authenticode signature. Moreover, they further processed the X.509 leaf certificates to extract information such as Subject CN, Issuer CN, PEM and DER hashes, validity period, signing hash, digital signature algorithm, signed file hash, and public key.

Prior studies on signed malware and PUP explored weaknesses that allow an adversary to sign malicious code. However, these weaknesses were limited to failures

in CA's identity vetting process and protection of issued signing keys. Bad practices and issues in the construction and issuance of code signing certificates have not been studied in a systematic way and consequently we do not know if Authenticode validation can defend against such potential issues. Similar to our goal, Kotzias et al. [24] highlighted this scenario that a timestamped certificate could remain vaild despite of being revoked and Kim et al. [23] highlighted security problems in revocation process such as CA's mismanagement of CRL and OCSP, and showed that it could not be prevented by Authenticode validation. We examine effectiveness of other steps of validation against flawed certificates as well as revocation checking.

2.2.2 TLS Certificates

Both TLS and Authenticode use public key infrastructure. Numerous TLS studies focused on the evaluation of PKI and digital signatures. In this section we also discuss prior works on the TLS certificates, since we are inspired by them for the analysis and testing of X.509 certificates.

Durumeric et al. [10] studied HTTPS certificate ecosystem uncovering problems that can undermine security of the ecosystem. They collected 42.4 million unique X.509 certificates from 109 million hosts by doing 110 exhaustive scans of the IPv4 HTTPS ecosystem during a course of 14 months. They used this dataset to study security questions related to certificate authorities and site certificates. Their investigation regarding the leaf certificates used by websites is similar to part of our work. They studied the public keys, the signature algorithms and the depth of these certificates. They highlighted that half of the trusted leaf certificates in their dataset used inadequately secure 1024-bit RSA key in their chain as well as the usage of broken MD5 as signature algorithm in April 2013 which is four years after publication of "MD5 Considered Harmful Today" [34].

Brubaker et al. [5] proposed Frankencert for adversarial testing of the logic behind certificate validation in different SSL/TLS implementations and libraries. They leveraged synthetic certificates to generate unusual test cases that are generated from combination of randomly mutated parts of SSL/TLS certificates. They tested functionality of seven different implementations/libraries and browsers while provided with these frankencerts. For interpreting the result of these tests they used differential testing. If one certificate was validated by one SSL/TLS implementation while

rejected by another implementation, the discrepancy was investigated to find potential flaw in individual SSL/TLS implementations. Diffential testing with 8,127,600 frankencerts uncovered 208 discrepancies.

The purpose of generating frankencerts was generating a set of test certificates carrying different combinations of attributes and extensions that were not seen in the existing certificates in the wild; however, they may be crafted into a certificate by an adversary. In other words, these frankencerts represented corner cases. Methods such as random fuzzing could not generate sound test cases, since it is unlikely that randomly generated strings could form a parsable certificate. Frankencerts are syntactically well-formed, however, they may violate the constraints required for a valid certificate. Thus, these certificates could verify if SSL/TLS implementations check these constraints properly. As frankencerts are parsable certificates that may violate the X.509 semantics, they could test code paths that were hardly executed and consequently identified potential flaws in certificate validation. These flaws could not be found by testing of normal certificates. We also leverage synthesized certificates to create unconventional test cases to explore proprietary implementation of the Authenticode validation. The Authenticode implementation have not been studied and tested as thorough as SSL/TLS implementations. Thus, our goal is gaining a clearer and more detailed view of the implementation along with exercising the implementation by certificates that are not commonly seen in the wild.

After Frankencert, numerous studies [17, 7, 36, 3] propose methods and tools to reveal a more comprehensive list of potential vulnerabilities. Kumar et al. [26] introduced Zlint tool to quantify HTTPS certificate misissuance in the wild. This tool reassessed the misissuance prevalence in the wild. Certificate authorities usually fail to adhere to applicable standards due to implementation errors or lack of concern while they are constructing certificates. Their tool tests certificates against policies set forth by the CA/Browser Forum Baseline Requirements and RFC 5280 and highlights the flaws. Their result showed that since 2012, misissuance has decreased significantly. In 2017, only 0.02% of certificates had errors. This improvement majorly caused by the large authorities that construct consistant certificates. However, misissuance and reported bad practices correlated with small authorities that construct non-conformant certificates.

In contrast, no prior work focuses on scrutinizing the code signing certificates and

evaluation of Authenticode validation. Our work shows that similar problems that have been reported for SSL/TLS certificates exist in issued code signing certificates and can lead to potential security vulnerabilities.

Chapter 3

Methodology

One main challenge for the measurement of the Authenticode is the lack of a comprehensive dataset of code signing certificates. Unlike SSL/TLS certificates, code signing certificates are not being logged in a centralized transparent way such as certificate transparency (CT). That is why it would be difficult to systematically collect a comprehensive list of the issued certificates and all the binaries that have been signed using them. The other challenge is the limitation that we have for studying Windows Authenticode as a proprietary product. Not only its source code is not available, its documentation provides limited explanation for the steps being checked for Authenticode validation. Thus, the order and details of the steps are not clear. Even studying of its functionality through testing and code analysis is challenging. How can we verify the output of our tests or interpret the test results? As the Windows code signing standard has only one proprietary implementation from Microsoft, methods such as differential testing that has been used to uncover the discrepancies in the SSL/TLS implementations and libraries [5] could not help us. For code analysis, the provided documentation of the APIs do not cover the details and the order of calling of the APIs for implementation of each step.

We tackle these challenges by designing two frameworks for data collection and analysis. To shed light on the issues of Windows code signing, we collect a dataset of signed applications distributed by third-party software publishers. We process the Authenticode signatures to extract attributes of code signing certificates. This information includes version, validity period, cryptographic algorithms, extensions, revocation distribution points, and issuers of the certificates as well as software publisher's

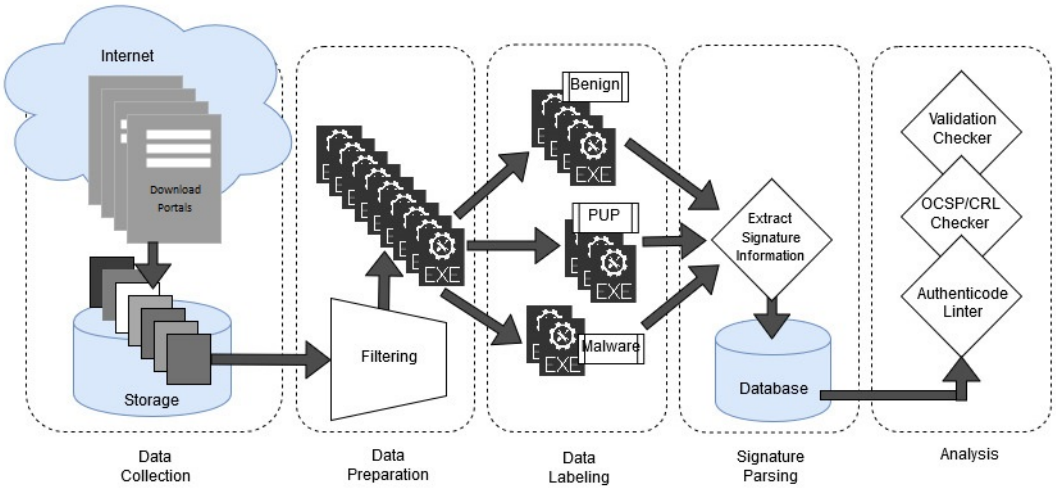

Figure 2: Data Collection and Analysis Pipeline.

URL and description. Then according to the Baseline Requirements for Issuance and Management of Publicly-trusted Code Signing Certificates [13], we investigate the extracted information to reveal weaknesses and flaws that can potentially be abused by adversaries.

3.1 System Overview

As illustrated in Figure 2, our data collection and analysis pipeline consists of five steps: application collection, input data preparation, data labeling, signature parsing, and Authenticode analysis.

Data Collection. The code signing certificates which were studied in the prior papers [22, 25, 23, 1, 24] were mainly extracted from malware and PUP. However, we intend to study the code signing certificates of supposedly benign binaries. One of the most important use cases of code signing certificates is assuring users about authenticity and integrity of third-party software that is downloaded from the web. Thus, for studying the effectiveness of the Authenticode mechanism in the wild, we target download portals as distributors of third-party software on the web. Targeting download portals as source of our dataset would allow us to understand if applications are being properly signed in the wild and whether in case of misissuance, if the Authenticode validation

can effectively prevent faulty signatures.

Input Data Preparation. The collected dataset of applications consists of different file types. As we intend to analyze signed executable files, we filter files with *.exe* extension as well as compressed files that might contain executable files, including *zip*, *rar*, *7-zip*, *gzip*, *tar* and *bzip2*. Part of the downloaded applications do not contain any executable files. They have these file types: *XML*, *ASCII*, *PDF*, *ELF*, *JPEG*, *Composite*, etc. Then we extract compressed files and search file directories recursively to locate all the executable files. We use Bash and Powershell scripts to automate this process.

Data Labeling. We use VirusTotal file scanning API to determine if a file is benign, potentially unwanted or malicious. Our collected dataset consists of supposedly benign files that are distributed on the web. So, first question that we aim to answer is how many files in our dataset are benign, malware, or PUP using the threshold-based approach that is proposed by Kwon et al. [27]. According to a study by Zhu et al. [39], the majority of the research studies (82 out of 93) that used VirusTotal for data labeling leveraged a threshold-based method. In line with a prior study [22], we use introduced metrics by Kwon et al. [27]: C_{mal} and r_{pup}. First metric shows the total number of VirusTotal engines that detect a file, and second metric determines the percentage of engines labeling a file as potentially unwanted. For calculating r_{pup}, labels that are indicators of PUP are filtered using the same set of keywords ("adware", "not-a-virus", "not malicious", "potentially", "unwanted", "pup", "pua", "riskware", "toolbar", "grayware", "unwnt" and "adload") utilized by Kwon et al. [27]. We consider binaries with $C_{mal} >= 20$ and $r_{pup} <= 10$ as malware and binaries with $C_{mal} >= 20$ and $r_{pup} > 10$ as PUP. We report benign files with a conservative threshold of $C_{mal} = 0$ (which occurs when no antivirus engine detects a binary as malicious) and with a less conservative threshold of $0 < C_{mal} < 20$.

We do not intend to filter only benign files for our analysis for two reasons. First, the purpose of this data labeling is to have more insight about the collected binaries. We leverage this information to compare signatures of benign applications with potentially unwanted and malicious applications to see how

conformant they are to the code signing requirements and specifications. Second, as Zhu et al. [39] discussed we cannot totally rely on the VirusTotal labels. Thus, we also analyze signed files that are labeled as badware.

Signature Parsing. We developed a tool to extract information regarding Authenticode signed executable files. Our tool first determines if an executable file is signed or not and then it will extract the signature's information from each signed binary and store it in a database. This information includes details about the issuers, validity, version and extensions of code signing and timestamping certificates. In this step we collect the required input for the analysis components. We use AuthenticodeExaminer [28] library to develop this tool.

Authenticode Analysis. First we measure how conformant are the issued Authenticode signatures and corresponding certificates to the baseline requirements. Then, we check if faulty signatures can pass Authenticode validation. For our analysis, we design and set up three components: Authenticode linter, OCSP/CRL checker, and validation checker. Our linter is a test suite that is specifically developed for Authenticode based on the specifications mentioned in Baseline Requirements for Issuance and Management of Publicly-trusted Code Signing Certificates. The OCSP/CRL checker determines if revocation distribution point is accessible for a code signing certificate. The OCSP/CRL test cases would verify if revocation distribution points are properly provided by the certificates and if these points are reachable. Eventually, the validation checker determines if Authenticode validation would prevent the potential violations. It is noteworthy that we execute the Authenticode linter and the validation checker on Windows 10 Pro version 1909 and the validation checker uses Signtool from Windows Kits 10.0.18362.0 . For testing the reachability of the CRL and OCSP servers, we use Wget command-line tool on Linux.

3.2 Framework Implementation

To tackle the existing challenges of studying Windows Authenticode, we design and implement two frameworks for data collection and analysis. We developed a crawling tool to collect a dataset of distributed applications on the web. Furthermore, our analysis framework results in the development of a certificate linter specifically designed

for Windows code signing. In this section, we discuss our design and implementation thoroughly.

3.2.1 Application Crawler

We redesign and implement a tool that is previously proposed by Rivera et al. [31]. This tool facilitates automatic downloading of Windows applications from a download portal, a website that provides software for download. Download portals generally follow similar design layouts; some of them list all the provided applications and some of them only list popular applications categorized based on platform, usage, or both. We target the most inclusive list of applications that each portal provides, and filter Windows applications. Regardless of usage or popularity of the software, we intend to collect all the applications offered by each portal. We choose eight out of 20 portals that have been crawled by Rivera et al. These are the top eight portals that during the course of our download provided the largest number of applications[31].

The automation tool that was proposed by Rivera et al. [31] is not publicly available. Thus, we need to redesign this tool. It is noteworthy that in the course of our redesign, we add a new functionality to our crawler so that it can monitor progress of downloading. We also improved error handling in our crawler so that our tool become more tolerant for network errors and can retry for the failed downloads. Our tool is in the process of release for public use of researchers.

Our tool sets up an iterable list of applications. We check the selected portals manually to determine the most inclusive application list that each of them provide. These lists are categorized based on the names of applications, popularity or usage (such as gaming, security tools, business, home and desktop, etc). Each category itself is comprised of several pages. Our tool locates all the web elements that represent categories; then it traverses all the pages under each category. For moving between pages, we check each portal manually to determine the web element that forwards to the next page. Portals usually provide a next/forward button. However, in some cases we extract a URL pattern that can be dynamically generated for corresponding page number that we intend to visit at each iteration.

On the other hand, some portals use one single page to present all the applications under each category and they would appear on the page dynamically as the page is

scrolled down. For these portals, our program scrolls down to the end of the page and then starts locating all the web elements that represent applications. These are parts of our program that require customization based on the design of a website.

Each application usually has its own page, presenting specifications and the corresponding download link. The reported size of each application is retrieved from its web page and stored in a database. We use this information to adjust a threshold. This threshold determines the waiting time that is needed for each download to be completed and refines functionality of our tool.

We use Selenium web driver,[4] a popular tool for automated web browsing, with the Chrome browser. Selenium is commonly used for automating web related tasks; however, it is not able to verify if download is completed successfully. When the driver sends the request, it does not have any built-in option to keep track of the status of the browser anymore.

We experienced download failures for similar reasons as enumerated in [31]. These reasons can be classified into two folds: Unreachability of the download link, or timeouts that can happen because of network issues on client or server ends. We leverage other Java libraries such as REST Assured, Jsoup and java.io to monitor progress of each download. First, we send a GET request and record the received response code in our database. If response code is 200, 301, 302 or 303, we expect it to be reachable. Thus, our tool will continue clicking on the link and initiating download. Receiving these response codes would not guarantee a successful download. We leverage a monitoring component on the file system. This component compares status of download directory before the start of each file download and after a specific threshold. This threshold will be calculated based on the file size and network speed. After reaching this threshold, our monitoring component will verify if download has completed successfully by checking the size and name of the last modified file in the directory. The last modified file should have recorded file size. Regarding the name, when Chrome browser downloads a file, it assigns an intermediate file extension "crdownload". If last modified file name has this extension, it means that download has failed or it is still in progress. We discriminate these two cases by monitoring the size of this file in one minute intervals. If size changes, it indicates the latter case and our monitoring tool would wait for the download to be completed. This method helps

[4] `https://www.selenium.dev/documentation/en/`

Algorithm 1: Data Collection from a Download Portal

```
Data: A download portal
Result: Downloading applications offered by a portal
Assigning URLs of categories to an iterable data structure;
for all the categories do
    for all the pages do
        for all the applications in each page do
            Retrieve platform of application;
            if couldn't retrieve platform of application then
                Insert URL and corresponding failure code into database;
            else
                if Platform is Windows then
                    if Download URL is not reachable then
                        Update database with corresponding URL and failure
                         code;
                    else
                        if Size of application is retrievable then
                            Update wait threshold based on size and network
                             speed;
                        else
                            Use default wait threshold;
                        end
                        Start download;
                        while download is in progress do
                            Wait;
                        end
                        Verify if download completed successfully;
                        Update database with corresponding URL and
                         success/failure code;
                    end
                else
                    Continue to next application
                end
            end
        end
    end
end
```

Portal	Alexa ranking Oct. 2019	Downloaded								
		EXE	***ZIP***	***RAR***	***7-ZIP***	***GZIP***	***TAR***	***BZIP2***	***OTHER***	***All Types****
uptodown	461	7333	3476	274	23	19	-	-	635	11760
softonic	227	9734	3499	150	18	11	-	3	755	14170
softpedia	2353	6461	3791	237	86	45	-	8	572	11200
tucows	19536	13611	2936	8	7	2	-	-	336	16900
freewarefiles	29272	3309	2813	69	32	9	-	6	281	6519
geardownload	893701	11253	4186	37	7	12	-	2	496	15993
bytesin	65866	1281	695	13	-	3	-	2	118	2112
soft112	4717	12477	12440	461	97	1955	11	264	264	27969
	total	65459	33836	1249	270	2056	11	285	3457	106623

*Not necessarily unique per portal.

Table 2: Applications Collected from Download Portals

our tool to tolerate sudden changes in our university's public network. Some portals do not report the size of the application or the reported size is not correct. For these cases, we define a default threshold (12 minutes). This threshold would work for a relatively large file. These cases were not observed frequently. Thus, the overall delay they cause would be negligible. An overview of our tool is indicated using Algorithm 1.

A summary of collected dataset of Windows applications are provided in Table 2. In total, 106,623 applications have been collected in August 2019.

3.2.2 Authenticode Linter

We design a linter that is comprised of a set of tests. The purpose of this tool is to identify the potential security breaches in an Authenticode signature. Potential errors may be made by the CAs upon issuing the code signing certificates or upon signing a binary (such as padding the signature). To identify potential errors, we codify the policies set forth by the Baseline Requirements for Issuance and Management of Publicly-trusted Code Signing Certificates. Our test cases assure that each leaf code signing certificate is conformant to these requirements. We also develop test cases based on known abuses that can potentially undermine the security of Authenticode. To develop this linter, we are inspired by AuthenticodeLint,[5] the only existing linter for Authenticode that provides 16 rules for testing a signed binary. Although these

[5]https://github.com/vcsjones/AuthenticodeLint

rules can provide insights regarding the security of a signature, they are not specifically developed based on the specifications. Therefore, they could not be sufficient for the aim of uncovering flaws based on the baseline requirements and policies. Not only we aim to uncover these issues, we study functionality of Authenticode validation toward them. We cover the MUST clauses of code signing specifications that correspond to ERROR severity level in standards documents. We also note that the baseline requirements have been continually evolving and some requirements are not retroactive, especially for Microsoft Authenticode that maintains compatibility with the older environments. As a result, an effective date has been maintained for the designed tests. Each test case is only applied to the certificates that were issued after the corresponding effective date.

3.3 Authenticode Analysis

As mentioned before, Authenticode has a proprietary implementation with closed source code and it has very limited documentation. Thus, the logic behind signing and validation processes are not clear. This makes analysis of Authenticode challenging. We tackle this challenge by designing test cases based on 1) known vulnerabilities of Authenticode and 2) policies set by the Baseline Requirements for Issuance and Management of Publicly-trusted Code Signing [13].

3.3.1 Baseline Requirements for Code Signing Certificates

In standards documents, failure to adhere to MUST clauses corresponds to errors. As we aim to evaluate functionality of Authenticode, we codify the requirements with MUST severity level that are maintained for signing certificates as a set of tests. If a certain test case fails, it can be due to an implementation error. We intend to predict potential issues using these test cases. However, we note that some issues are the intentional design features of Authenticode so that it maintains compatibility. Therefore, passing each test case depends on a maintained effective date. We do not expect certificates for conformance to a specific requirement, if they were issued prior to the corresponding effective date. In the following we describe our designed set of tests for analyzing Authenticode and discuss the potential security issues.

Certificate Version. Certificates must be of type X.590 version three. The major difference between version two and three is the addition of certificate extensions. Some of these extensions such as Key Usage, Extended Key Usage and Basic Constraint play a critical role for PKI security and consequently trust that Authenticode aims to maintain. Moreover, old certificates especially root and intermediate ones with version one or two still exist in the wild. Thus, checking of the certificate version is required.

Strong Hash Algorithm. Using broken hash algorithms such as MD5 or SHA1 can break security of the Authenticode signature. Flame malware is an example that used chosen-prefix collision attack to produce a counterfeit version of a legitimate certificate issued by Microsoft. Another abuse scenario is exploit of a MD5 or SHA1 collision attack on a legitimate signed binary. An adversary can copy signature of a legitimate software and append it to his malicious code. As it has the same digest, the appended signature would be valid.

Requirements for Basic Constraints. This extension determines if a certificate is authorized to act as a CA certificate or as an end-entity certificate. Thus, checking of this extension is required for code signing certificates. Otherwise, an adversary can use an unauthorized certificate for issuing Authenticode signature and break Authenticode trust. In a prior TLS study [5], violating cases have been reported. Thus, we implement test cases for examining the presence of duplicate extension as well as proper setting of CA field for intermediate and end-entity certificates.

Presence of Certificate Policies. This extension must be present in all the signing certificates. It indicates the policies that are followed by the issuer for certificate issuance as well as other operational practices such as revocation.

Requirements for Key Usage. This extension defines authorized usages of a certified public key. Thus, its presence is critical for validation. We examine bit settings of this extension that are specifically required for code signing to be set to *DigitalSignature*. Defined purposes by Key Usage and Extended Key Usage must be consistent [6].

Requirements for Extended Key Usage. This extension defines purposes

that can be used by certified public key in addition or instead of purposes determined in the key usage extension. This extension must contain specific value of *id-kp-codeSigning* for code signing certificates and must not contain *anyExtendedKeyUsage* or *serverAuth* value which is intended for TLS WWW server authentication [6]. If a violation of these requirements could not be caught by validation process, a TLS or timestamping certificate (for example) can be abused for signing applications. This will break the authenticity assurance that Authenticode is supposed to maintain; considering that the vetting process that is required for obtaining a TLS certificate is not the same as the one required for Authenticode.

Strong Public Key Algorithm. Since all the valid code signing certificates in our dataset used RSA algorithm, we only verify the minimum required key size for RSA which must be 2048 bits.

Requirements for CRL Points. CRLDistributionPoint is the extension maintained for providing CRL access point. Presence of this extension is optional for code signing certificates. However, if it is present, there are specific requirements that are obliged to be met. It must not appear as a critical extension. Moreover, the HTTP URL of the CRL service for corresponding certificate authority must be provided.

Requirements for OCSP Points. AuthorityInformationAccess is the extension maintained for providing OCSP access point. This extension must be present and must be set to critical in code signing certificates. The extension must contain HTTP URLs of the corresponding certificate authority's OCSP responder as well as HTTP URL of the root CA's certificate. We examine all the signed binaries to see if the access points for revocation information are provided properly.

Validity Period. Validity of a certificate starts from the issue date of a certificate and ends on the expiry date of the certificate; determined by the two specific fields of notBefore and notAfter. Maximum validity period for code signing certificates must not exceed 39 months.

Timestamped Signature. Timestamping is an optional extra step in the code

signing process. When a signed application is timestamped, its signature will be preserved permanently, unless it gets revoked for a specific reason such as private key compromise. If a signed application is not timestamped, its signature will become invalid upon expiration of its corresponding code signing certificate. For software authors, it will not be easy to recollect the distributed software among users and renew their signatures. Considering the distributed nature of code signing, timestamping is a worthwhile option for software authors. We examine the prevalence of timestamped signatures in our dataset.

3.3.2 Complementary Adversarial Testing

We developed a set of test cases according to the code signing requirements and known vulnerabilities. These test cases reveal issues in the code signing certificates that are collected from the web. However, we also intend to exercise the code paths of the Authenticode validation that would not be executed normally. In other words, we want to test corner cases that may not appear in the wild. Therefore, we customize the Frankencert tool that is proposed by Brubaker et al. [5] for the adversarial testing of the SSL/TLS implementations and libraries. Frankencert generates synthesized certificates that may be completely different from the conventional certificates. This tool gets a set of certificates as seeds and break them down into parts. Then, it will mutate random combinations of these parts so that they satisfy the ASN.1 grammar for X.509. These synthesized certificates will be parsable. But, they may violate the X.509 semantics and include unconventional combinations of extensions, extension values, critical and non-critical flags for the extensions, etc.

Synthesized Code Signing Certificates. We use our corpus of certificates as seeds and generate 500,000 certificates. These synthesized certificates are made from randomly mutated parts of the real certificates of our dataset. Application authors do not have to use only Microsoft-provided signing tools such as Signtool [8]. We use both Signtool and OsslSignCode [4] to sign 500,000 dummy executable files and we provide the synthesized certificates to both of these tools. OsslSignCode accepts all the certificates for signing the executable files. However, Signtool only accepts 294,255 certificates for signing and after applying the signature by Signtool, only

55,453 files could pass the Authenticode validation. This indicates that the minimum requirements checked by Signtool during the signing process are not the same as the minimum requirements for validation. A binary does not have to be signed by a Microsoft-provided signing tool, necessarily. So, there is no guarantee that a third-party signing tool does a thorough check for the code signing requirements and issues a sound signature. Although in this test the Authenticode validation prevents 81.15% (238,802/294,255) of the potentially faulty certificates, yet we cannot be sure that the Authenticode validation can effectively prevent all the possible flaws.

We find out 719 certificates in our dataset that are not approved by Signtool for signing, however, they could pass the Authenticode validation. Unlike similar approaches, we could not inspect the closed source code of Authenticode to identify the root causes of the rejected certificates. We could not use differential testing to find discrepancies in certificate validation due to the fact that Windows code signing has only one implementation from Microsoft. Chktrust [18] is an open source implementation for Authenticode. But, Chktrust yields less accurate results compared to Microsoft-provided implementations such as Signtool and Sigcheck [35]. Thus, we did not find its test results insightful or reliable. As a result, we could not leverage differential testing for interpreting the test results or identifying discrepancies. Yet, we analyze these 719 validated certificates and report the found discrepancies based on the baseline requirements for code signing.

With the help of our test cases, we found violations that are only seen in the invalid certificates of our dataset. The reason for validation failure in these violating certificates does not originate from the found violation itself. As the detailed steps of Authenticode validation and the order of them are not clear, it is likely that some issues are hidden behind some other errors. For example, the violations that are only seen in the self-issued certificates are deemed invalid because of an untrusted root error. There is a possibility that there are maintained checks for these specific violations. However, other errors such as an untrusted root have occurred earlier in the process of validation checking. Hence, the validation process exits with an error code before reaching to that specific check in the implementation. Therefore, we could not verify if the Authenticode validation can effectively prevent such violations.

We use these synthesized certificates to see if the aforementioned violations can pass the Authenticode validation.

Chapter 4

Results

4.1 Summary of the Input Data

We start by providing information about our input data and the effect of the preparation step on our dataset. As depicted in Table 2, we collected 106,623 applications from eight download portals. After removing duplicate applications and extracting executable files, we have 79,128 distinct applications. Other than duplicate files, some of the archive files could not be decompressed successfully or do not contain executable binaries.

4.2 Signed Applications in the Wild

In this section we want to understand the current status of code signing in the wild.

4.2.1 Overview

The collected dataset is a sample of freeware offered by the third-party software publishers on the web. Among our dataset, 35.11% (27,789/79,128) of these applications are signed. Our result shows that compressed applications have a relatively smaller share among the signed applications (13.05% (3,629/27,789)). These applications have compressed data types including *zip*, *rar*, *7-zip*, *gzip*, *tar* and *bzip2*. However, 86.94% (24,160/27,789) of the applications are not compressed and have *.exe* file type.

Portal	**Count (%)**
filecluster	1071/1741(61.51%)
geardownload	5355/12174 (43.98%)
softpedia	3408/8732 (39.02%)
soft112	5853/15222 (38.45%)
softonic	3753/11610 (32.32%)
uptodown	2507/8598 (29.15%)
tucows	4363/15487 (28.17%)
freewarefiles	1479/5564 (26.58%)
Total	27789/79128 (35.11%)

Table 3: Number of Signed Applications Distributed by Download Portals

Table 3 shows that 61.51% of the applications distributed by Filecluster are signed which is the largest among the other seven portals. Bearing in mind that Filecluster is offering the least number of applications, this is due to a relatively less variety in the offered applications. Besides, these portals do not require publishers to sign their applications. Thus, differences in the number of signed applications per portal can result from differences in the software authors that distributed their applications through these portals.

	Signed Applications				*Unsigned Applications*			
Portal	*Benign* ($C_{mal} = 0$)	*Benign* ($0 < C_{mal} < 20$)	*PUP*	*Malware*	*Benign* ($C_{mal} = 0$)	*Benign* ($0 < C_{mal} < 20$)	*PUP*	*Malware*
geardownload	3912	1468	96	0	4602	3078	21	1
filecluster	813	259	2	0	353	326	8	4
softonic	2674	1019	26	1	4782	3154	41	11
softpedia	2540	829	37	0	3046	2386	48	14
freewarefiles	1190	290	3	0	2658	1573	4	1
uptodown	1594	849	59	0	3498	2570	81	17
soft112	4101	1419	108	0	5584	3820	26	3
tucows	2620	1586	154	3	5389	5679	111	12
Total(Unique)	16200	6478	458	4	25757	19815	329	62
Verified Signature	14719	5323	380	4	-	-	-	-

Table 4: Distribution of Signed Malware/PUP in the Wild

Our next question is regarding the scope of code signing abuse in the wild. In other words, we want to measure the prevalence of distributed signed badware (PUP and malware). Prior work [38, 1, 24, 22] have studied abuse of code signing by both of the malicious and potentially unwanted applications. However, their datasets are mainly

biased on the suspicious binaries. We provide quantitative numbers on maliciousness of a subset of supposedly benign applications that are distributed on the web.

As depicted in Table 4, 39.37% (22,678/68,250) of the benign applications are signed. 88.37% (20,042/22,678) of their signatures are validated successfully. Also, 58.19% (458/787) and 6.45% (4/66) of suspicious files are signed with respectively potentially unwanted and malicious applications. The number of signed suspicious files shows that ill-intentioned publishers abuse signatures to make their badware seem more legitimate. It is also worth noting that the number of signed suspicious binaries for Geardownload, Soft112, and Tucows portals are more than the number of unsigned suspicious files. 82.96% (380/458) of PUP and 100% (4/4) of malware carry valid signatures. As results show, number of malicious files in our dataset is limited.

Problem	**#Occurrences**	**#Distinct Certificates**
Expired	42	23
Revoked	8	7
Bad Digest	6	1
Malformed Signature	6	3
Total	62	34

Table 5: Observed Problems with Signatures of PUPs/malwares

4.2.2 Verification Errors

We take a closer look at the problems of unverified signatures for both benign and suspicious applications. 1,434 applications carry invalid certificates as indicated in Table 7. Expired certificate is the most common issue observed for 812 applications. 78 of them get this error, although they are timestamped. Checking validity and signing times of the timestamp certificates shows that these signatures were timestamped, although the code signing certificates were expired. However, there are cases that the signatures were timestamped while the code signing certificates were valid and we could not identify any problem with their validity periods. Figure 4 in appendix A indicates a screenshot of an example certificate. The error message claims that a

required certificate is not within its validity period, although all the certificates in the chain are valid. These error messages that are not clear and relevant would dissuade users from using a program and consequently impact the usability of this security mechanism. Furthermore, if an error message is ambiguous, users will not be able to find the reason of the error. For example, the Crypto policy error (presented in Table 14, Appendix A) does not contain relevant and specified information for the users. Consequently, users would not be able to do anything to resolve the problem and it would impact the experience of the product negatively. Moreover, the worst case for this security product is that users would bypass the prompt message and proceed to the installing or launching of the application.

			Validation Error						
Issuer	*Valid* Signature	*Invalid* Signature	*Expired*	Revoked	Chaining	UntrustedRoot	BadDigest	CryptoPolicy	MalformedSignature
Comodo	8742 (28.89%)	677 (21.62%)	343	250	1	53	29	0	1
Symantec	4263 (14.08%)	240 (7.66%)	224	6	0	0	10	0	0
Verisign	4185 (13.83%)	582 (18.59%)	435	35	20	1	91	0	0
DigiCert	3599 (11.89%)	130 (4.15%)	100	18	3	0	9	0	0
Thawte	2304 (7.61%)	250 (7.98%)	69	144	12	2	20	1	2
GlobalSign	2017 (6.66%)	159 (5.07%)	85	52	2	0	17	0	3
Usertrust	1411 (4.66%)	172 (5.49%)	149	0	0	0	23	0	0
Microsoft	1068 (3.52%)	12 (0.38%)	2	0	0	6	6	0	0
Sectigo	788 (2.60%)	0 (0%)	0	0	0	0	0	0	0
Go Daddy	516 (1.70%)	42 (1.34%)	36	2	0	0	4	0	0
StartCom	450 (1.48%)	168 (5.36%)	121	45	0	0	2	0	0
Certum	405 (1.33%)	24 (0.76%)	21	0	0	0	3	1	0
WoSign	324 (1.07%)	15 (0.47%)	9	5	0	0	1	0	0
Starfield	51 (0.16%)	8 (0.25%)	7	0	0	0	1	0	0
SSL.com	12 (0.03%)	0 (0%)	0	0	0	0	0	0	0
Total	30258	3130							

Table 6: Top known Issuer Organizations

Another problem with the invalid signatures is using malformed signature that is only observed for the suspicious applications. Problem of a malformed signature originates from a vulnerability in the Authenticode validation. This vulnerability allows injection of data into an Authenticode signature without invalidating it. Microsoft

resolved this problem in security advisory 2915720 [20]. Stricter Authenticode validation, implemented with MS13-098, was planned to be enabled from June 10, 2014. But, later in July 29, 2014, they decided to disable enforcement of the default stricter verification behavior in the supported releases of Microsoft Windows, and they made it available as an opt-in feature. It is mentioned in the *Wintrust* documentation that setting the *EnableCertPaddingCheck* registry key under the *Wintrust* config enables stricter verification. We could not find this registry key in Windows 10. So, we get this specific error for six applications without changing the default settings which is not in compliance with the Microsoft documentation.

In contrast, our linter reveals this discrepancy that 24 applications in our dataset carry valid padded signatures. This fact that none of these applications are benign implies that a legitimate publisher would not risk his application to be leveraged by malware. These padded signatures can be used as a heuristic to identify abuse of digital signatures.

Problem	**#Occurrences**	**#Distinct Certificates**
Expired	812	383
Untrusted Root	305	213
Revoked	192	62
Chaining	109	51
Bad Digest	8	3
Crypto Policy	7	6
Test Root	1	1
Total	1434	716

Table 7: Observed Problems with Signatures of Benign Applications

Table 6 summarizes the statistics of the issued Authenticode certificates by the top known certificate authorities. We categorize invalid certificates of each organization based on the validation error. Expired certificates are the most prevalent reason for the validation failure. The second common error is a revoked certificate. Numbers of occurrences of both expired and revoked certificates are high for StartCom issuer compared to the total number of signatures that are observed from this issuer in our dataset. In contrast, Symantec has respectively low numbers of expired and

revoked certificates. The other validation errors such as *MalformedSignature*, *BadDigest*, *Chaining* and *UntrustedRoot* do not relate to the issuers. Bad practices or abuses conducted by software authors would result in these errors.

4.2.3 Publisher Information

SpcSpOpusInfo in the Authenticode structure is maintained for the publishers to specify the program name, the description, and a website URL containing more information about the signer. 14,892 applications in our dataset have not provided proper information regarding the publisher. 87.81% (13,077/14,892) of them carry valid signatures. 55.86% (8,319/14,892) do not provide any name or description and 61.17% (9,110/14,892) do not provide any URL. 0.03% (519/14,892) of the accompanying URLs are not valid URIs; "`www.bitdefender.com`", "`iMagicPtyLtd`", "`http://$(LANG_CIMAWARE_DOMAIN)/main/products/deletefixphoto.php`", "`http://`", "`/windowsxp/home/downloads/bliss.asp`" and "`www.edrawsoft.com.`" are examples of invalid entries. 49.92% (7,435/14,892) do not provide any name, any description or any URL.

4.3 Specifications Violations

Another question that we want to answer is if applications are being signed properly. If not, the validation process will defend against the violations? To answer these questions, we identify potential violations in the wild by examining the signatures to see if they conform to the baseline requirements for code signing. Our authenticode linter codifies these requirements and policies. Our linter's test cases determines the specifications that are violated in the wild and our validation checker component tests if the validation process would prevent the violations.

First version of the code signing baseline requirements was published in September 2016. Microsoft has added the requirement to use these specification as a new standard by February 1, 2017. We specifically present results of our test cases after this date to demonstrate effectiveness of this document, as well. As depicted in Table 8, the number of observed problems in the wild significantly decreased following that

date. Yet, the existence of a valid signature that is created using a faulty certificate implies that the validation process does not take specific checks into account.

					*February 2017**			
Problem	*Valid*		*Total*		*Valid*		*Total*	
	#occurrences**	#certs***	#occurrences	#certs	#occurrences	#certs	#occurrences	#certs
certificatePolicies is not present	1195	620	1785	971	43	15	116	58
basicConstraints is present, but *CA* field is set true	0	0	14	9	0	0	0	0
keyUsage is not present	967	574	1500	886	43	15	106	47
keyUsage is not marked critical	184	13	223	31	3	3	6	5
For *keyUsage*, *CRLSign* bit positions are set	0	0	2	2	0	0	0	0
For *keyUsage*, *keyCertSign* bit positions are set	0	0	4	3	0	0	0	0
For *keyUsage*, *digitalSignature* bit positions are not set	967	574	1501	887	43	15	107	48
For *extKeyUsage*, the value *id-kp-codeSigning* is not present.	67	37	398	269	1	1	61	29
For *extKeyUsage*, the value *serverAuth* is present.	0	0	1	1	0	0	2	2

* Effective date.

** Depicts number of distinct applications signed by violating certificates.

*** Depicts number of distinct violating certificates.

Table 8: Specification Violations Found in the Wild

4.3.1 Certificate Version

We observe 18 distinct applications carrying version one code signing certificates. These applications are signed by 11 distinct certificates, however, none of them are valid and they are not issued by any known certificate authorities. Their validation fails due to the chaining error or the untrusted root. Yet, it is worth mentioning that version one code signing certificates in our synthesized corpus of data could

successfully pass the validation, which implies that the version field is not checked for Authenticode validation.

This question is raised that version one certificates can skip the checking of the critical extensions such as the Basic Constraints. We design a specific test case to validate a version one certificate that does not have any Basic Constraints extension. If a certificate has the Key Usage extension and the Extended Key Usage extension defined for code signing and does not have the Basic Constraints extension, it cannot pass the validation. However, if a certificate does not have any Key Usage extension, any Extended Key Usage extension and any Basic Constraints extension, it can pass the validation. Based on this test, we conclude that the Authenticode validation expects all the certificates to be of version three and regardless of the certificate version, the checking of the Basic Constraints extension depends on the Key Usage and the Extended Key Usage extensions.

4.3.2 Certificate Extensions

The majority of our dataset are signed by version three certificates. The important characteristic of version three certificates is presence of extensions. Besides, from technical point of view, difference between code signing certificates and TLS certificates originates from differences in bit settings of the Key Usage and the Extended Key Usage extensions. Thus, we find it essential to examine the requirements that are specifically set for the extensions of code signing leaf certificates.

Certificate Policies

The number of issued certificates that do not carry this extension has decreased since 2017. Yet, we observe 15 valid certificates issued by Microsoft in 2017, 2018, and 2019 that do not contain this extension. The number of valid signatures indicates that this extension is not checked for the Authenticode validation.

Basic Constraints

If this extension is present in a leaf certificate, the CA field must be set to false. For all the CA certificates that contain public keys used to validate the digital signatures on the certificates, the CA field must appear as critical and must be set to true. Thus, a leaf code signing certificate with a CA bit set to true is invalid. CA bit is set to true in nine certificates of our dataset. However, none of them could pass validation. Eight of them are self-signed certificates and the corresponding validation error message explains that *"A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider"*. The other certificate is not self-signed, however, it does not contain the intermediate CA certificate. Thus, it throws this error: *"A certificate chain could not be built to a trusted root authority"*.

First error message received for the self-signed certificates cannot verify if Authenticode validation maintains any specific check for this extension or not. For the latter case, the error message shown through the Windows file explorer indicates that *"The issuer of this certificate could not be found"*. So, apparently the failure originates from the chain building. For further investigation of the maintained checks for this extension, we use our synthesized certificates. We could verify that a leaf code signing certificate cannot be validated if its CA field is set to true. However, our investigation reveals three other violations: presence of duplicate extensions, not critical extension for the intermediate certificate, and version one certificate without any extensions. These three violations will not cause the validation fail which is a semantic error that can be leveraged by an adversary to bypass the crucial requirements related to the Basic Constraints extension.

Prior work [3] reported the first violation as a security vulnerability for TLS certificates. This vulnerability can be security critical for certificates that have two Basic Constraints extensions; one with the CA field set to true while the other one has its CA field set to false. This can allow an end-entity certificate to take action as a rogue CA. Second violation is the missing critical flag which may allow a platform (for example the Authenticode validation) to disregard the Basic Constraints extension if its value is not recognizable. This can lead

to improper checking of certificate's authorization. The third violation can be security critical as well. According to the baseline requirements, only version three certificates should be used for code signing. Nevertheless, version one certificates that do not have any extensions could be validated successfully regardless of the authorization check.

Key Usage

This extension must be present to ensure that a certificate is authorized for its purpose. In our corpus of data, 574 valid certificates that were issued before February 2017 do not contain this extension. These certificates are issued by Microsoft, Thawte, and Centrum. 15 certificates with the same problem were issued after this date by Microsoft.

This extension must be marked as critical. 13 valid certificates issued by Intel, Dell, Cetrum, ACNLB, and Verisign in 2001, 2008-2011, 2014, and 2016 violate this specification. Three valid certificates issued by Intel in 2018 have the same problem.

Regarding the bit positions for the Key Usage extension, the *digitalSignature* bit must be set. This bit position in one certificate in our dataset is not set. The Key usage of mentioned certificate is *nonRepudiation*. This certificate could not pass validation since its issuer is not trusted. The signature properties in the Windows file explorer shows no problem corresponding to the invalid Key Usage Extension. In other words, a certificate that is not authorized for code signing can pass Authenticode validation. We further verified this using our synthesized certificates. There are some certificates having their Key Usage extension set to *CrlSign* and *KeyCertSign* and their Extended Key Usage extension set to *TimeStamping* that could successfully pass the Authenticode validation.

We also observe a self-signed certificate issued in 2018 with *none* value for the key usage. Properties field of the certificate shows this error message: *"This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store"*. Furthermore, the other 574 valid certificates issued before February 2017 and those 15 certificates that are issued

after this date, do not contain this extension either. Among our synthesized certificates, we have cases that do not contain any Key Usage extension and their Extended Key Usage extension is only set to *TimeStamping*, and they could successfully be validated. This indicates that Key Usage is the other extension that is not checked properly for the Authenticode validation.

The bit positions of *keyCertSign* must not be set for the Key Usage extension as well. Three certificates have their *keyCertSign* bit positions set. One of them is not verified because the integrity of the signature could not be verified. The second certificate is self-signed and is not verified because it terminates with an untrusted root. The third certificate is not self-signed, however, it does not contain its issuer certificate. It is noteworthy that the CA bit for this certificate is set to true, although it is an end-entity certificate. However, the validation failure does not originate from this problem nor the problem with the Key Usage extension. We observe a similar problem for the *CRLSign* bit positions in the Key Usage extension. These bit positions must not be set either, however, 133 certificates in our synthesized dataset are valid in spite of having their *keyCertSign* and *CRLSign* bit positions set.

Extended Key Usage

This extension is required for code signing certificates and must be set to *id-kp-codeSigning*. The numbers reported in the eighth row of Table 8 indicate that not all the certificate authorities are conformant to this specification. However, according to the Wintrust library documentation, certificates with no *Extended Key Usage* are also accepted by the `WINTRUST_ACTION_GENERIC_VERIFY_V2` policy.[6] This question is raised that how verification process checks if a certificate is authorized for code signing. For example, we observe three valid signatures that are issued by certificates from Certum in 2007-8 without any Key Usage or Extended Key Usage extensions defined. Furthermore, there are cases in our synthesized certificates that do not contain any Key Usage extension and the Extended Key Usage extension exists, however, it is not set to the

[6] `https://docs.microsoft.com/en-us/windows/win32/seccrypto/example-c-program--verifying-the-signature-of-a-pe-file?redirectedfrom=MSDN`

id-kp-codeSigning. This implies improper checking of the Extended Key Usage extension as well.

Three certificates in our dataset have *serverAuth* value present for the Extended Key Usage extension, however, their corresponding signatures are not validated successfully. The validation failure originates from an untrusted root and problem with the chaining since the issuer's certificate is not appended. For the further investigation of the maintained checks for this extension, we use our synthesized certificates. Our investigation reveals that Authenticode validation does not prevent leaf certificates that have the *serverAuth* value present for their Extended Key Usage extension.

4.3.3 Cryptographic Algorithms

Code signing baseline requirements enforce cryptographic constraints for the digest algorithm, RSA, DSA and ECC curve. Some of these algorithms such as MD5 and RSA with 512-bit and 1024-bit key sizes were cracked several years before the publication of the Code Signing baseline requirements. We report our results before and after the chosen effective date of Microsoft for the first version of the baseline requirements which is February 1, 2017.

Public Key Algorithm

All the code signing certificates in our dataset used the RSA algorithm. The minimum required key size for RSA is 2048 bits. As indicated in Table 9, all the valid signatures issued after February 2017 meet this requirement and used 2048-bit and 4096-bit key sizes. We also observe 43 distinct applications carrying 15 distinct self-signed signatures that used 1024-bit key size. Thus, our results show that the legitimate code signing certificates that are issued by known certificate authorities satisfy the minimum security requirements for the public key algorithm. On the other hand, we observe that the Authenticode validation does not reject the weak key sizes. 1741 and 11 distinct applications respectively used 1024-bit and 512-bit key sizes and their signatures validated successfully.

Digest Algorithm

So far, we discussed the violations that would affect the authenticity assurance of Authenticode. Now, we want to talk about the integrity assurance that Authenticode provides. Hash algorithm is used in several places in the process of signing a file. As we discuss the integrity assurance of a software, we are referring to the hash algorithm that is used for calculating hash of a code. However, same vulnerability exists for the hash algorithms that are used by the corresponding certificates.

			*February 2017**	
Algorithm Key Size	#occurrences**	#certs***	#occurrences	#certs
RSA 4096	83	38	76	24
RSA 2048	10049	3823	5681	1646
RSA 1024	1741	916	0	0
RSA 512	11	10	0	0

* Depicts the effective date for the baseline code signing requirements.

** Depicts number of distinct applications carrying valid signatures.

***Depicts number of distinct valid signatures.

Table 9: Observed Public Key Algorithm and Key Size in the Wild

If a hash algorithm is broken, the signed hashes could be abused by malicious code authors. Both MD5 and SHA1 are broken hash algorithms. MD5 is deprecated by Microsoft in security advisory 2862973 [12] which has been effective since February 2014. According to this announcement, Microsoft will allow the signed binaries that were signed before March 2009 to continue to work even if the signing certificate used the MD5 hash algorithm.

As indicated in Table 10, 667 applications in our dataset carry a valid signature that used the MD5 digest algorithm, although they were issued after March 2009. We also report the number of signatures that applied this weak

hash algorithm after 2017 for two reasons: To demonstrate the recent application of this broken hash algorithm by software publishers and to bring into attention the fact that validation process do not prevent the application of a weak digest algorithm for the digital signatures. These certificates are issued by Comodo, Thawte, Symantec, GoDaddy, and Sectigo.

	Effective date			
	After March 2009		*After January 2017*	
Hash Algorithm	#occurrences*	#certs**	#occurrences	#certs
MD5	667	290	94	28
SHA1	7711(2)	2860(2)	4775(11)	1363(3)
SHA256	650(1080)	193(294)	1456(2096)	519(573)
SHA384	0	0	1(0)	1(0)
SHA512	0	0	4(1)	3(1)

* Depicts number of distinct applications carrying valid signatures.

**Depicts number of distinct valid primary signatures (secondary signatures).

Table 10: Digest Algorithms Used in the Wild

Collision and chosen-prefix attacks are now practically feasible on SHA1 [21]. In other words, even a limited academic budget can afford the required resources and cost for breaking SHA1. Microsoft has announced a plan since 2015 [29] for phasing out this protocol, however, this plan left code signing unaffected. We report quantitative numbers regarding the usage of SHA1 in the wild. Dual-signing is a solution proposed for supporting older versions of Windows. Our results indicate the number of primary and secondary signatures that use MD5/SHA1/256/384/512 algorithms.

Among 94 applications that used MD5 digest algorithm (after January 2017) for their primary signature, 14.89% (14/94) of them are dual signed using SHA256 digest algorithm. Likewise among 4,775 applications that used SHA1 digest algorithm (after January 2017) for their primary signature, 43.12% (2059/4775) of them are dual signed using SHA256 digest algorithm and one of them is dual signed using SHA512 digest algorithm.

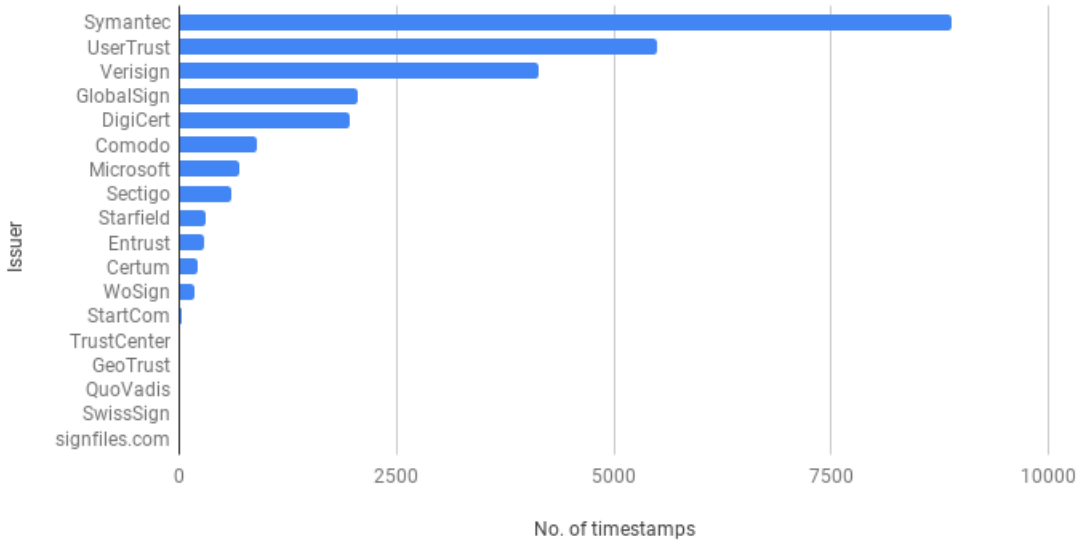

Figure 3: Observed Timestamp Authorities in the Wild

4.3.4 Certificate Validity Period

Another requirement that has been overlooked about certificates is their validity period. A code signing certificate issued to a subscriber or signing service must not have a validity period of more than 39 months. In our dataset 539 applications are signed by 121 distinct certificates with the validity period of more than 39 months. These are the applications that have been signed after February 2017 and are validated successfully in spite of violating this requirement. These certificates were issued by Comodo, Symantec, Sectigo, DigiCert, Go Daddy, and GlobalSign.

Contained HTTP URL	**#certs***
Both CRL and OCSP	7075/7398
OCSP only	11/7398
CRL only	301/7398
No OCSP	312/7398
No OCSP and no CRL	11/7398

*Depicts number of distinct certificates.

Table 11: OCSP/CRL in Valid Certificates

After expiration of a code signing certificate all the signatures that were issued by this certificate would become invalid unless they have been timestamped during the validity period of the certificate. 87.56% (20,153/23015) of the distinct applications of our dataset have timestamps. 95.90% (19,327/20,153) of these applications carry a valid signature. As indicated in Figure 3, Symantec, UserTrust, Verisign, GlobalSign, and Digicert are the top five timestamping servers that have been frequently used by our signed binaries.

Error	**#Occurrences***	**#URLs****	**Issuer[s]**
Name or service not known	6084	9	Verisign, AOL, Starssl, Safescrypt, Whosign, Godaddy, SSL.com, Swisssign
Read error connection reset by peer in headers	2027	2	Trustcenter, Globaltrust
Error 404 not found	1638	1	Globalsign
Error 403 forbidden	2909	2	Microsoft
Error 530 no description	279	1	Globalsign
Error 503 no healthy IP available for the backend	15	1	Globalsign
Error 503 service unavailable	9	2	Globasign, Swisssign
Error 503 first byte timeout	7	5	Whosign, Globalsign
Error 522 no description	2	1	Globalsign
Error 503 Timed out while waiting	3	3	Globalsign
Error 502 bad gateway	1	1	Globalsign
Error 520 no description	1	1	Globalsign
Error 500 internal server error	1	1	Swisssign
Total	12976	30	

*Depicts total number of occurrences for the corresponding problem.
**Depicts number of distinct CRL URLs that are observed having the corresponding problem.

Table 12: Observed Errors for Unreachable CRL URLs

The next requirement that plays an important role in maintaining the effectiveness of Authenticode is the proper implementation of revocation checking. Prior work [23] has studied the effectiveness of revocation as the primary mitigation mechanism against abusive usage of code signing certificates. Kim et al. [23] emphasized on the

importance of the effective and prompt revocation of abusive certificates and dissemination of the revocation information. However, we want to show that even if this information is maintained properly by certificate authorities, proper application of this information in the Authenticode validation is determinant. Authenticode validation applies a soft-fail revocation checking policy. So, if the revocation information is not accessible for any kind of reasons, the certificate will be trusted regardless of its actual revocation status.

The revocation information of a certificate is provided through the CRL or OCSP method. From a technical point of view, this information is presented to platforms using the *CRLDistributionPoint* or *authorityInformationAccess* extension of a certificate. According to the specification [13], *CRLDistributionPoint* may be present, but *authorityInformationAccess* must be present in a code signing certificate. Both extensions must contain the HTTP URL of the CRL/OCSP service of the corresponding CA. Furthermore, *authorityInformationAccess* must contain HTTP URL for the root CA's certificate. Seven valid certificates in our dataset do not contain the HTTP URL of their root CA. These certificates are issued by Thawte, Verisign, Usertrust, Certum, and America Online in 2003, 2004, 2007, 2008, 2010, and 2011. 17 valid certifiactes contain the HTTP URL for the root CA's certificate, however, they do not provide the URL of their CA's OCSP responder. These certificates are issued by GlobalSign, WoSign, GeoTrust, Microsoft, Dell, Intel, ACNLB, and RBC Hosting Center in 2006-2012, 2014, 2015, and 2019.

70 applications in our dataset are signed since February 2017 using 38 distinct certificates that do not contain any OCSP or CRL extensions. None of them are validated. Verification errors are due to the fact that they are self-signed. So, they are caught by an untrusted root or failed chaining (because of not containing a trusted CA certificate). No recent misissuence by the certificate authorities is observed in our dataset. According to the documentations of the Crypto API,[7] revocation checking will be done if the chain building successfully terminates in a trusted root certificate. Thus, these self-signed samples cannot aid us to verify if the Authenicode validation takes revocation information into account effectively or not. However, it is noteworthy that 534 applications in our dataset are signed by 312 valid certificates before 2017 that have no OCSP URL provided (depicted in Table 11). 11 valid certificates are

[7] `https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/ee619754(v=ws.10)?redirectedfrom=MSDN`

issued by Thwate in 2000-2003 and do not contain CRL extention either.

Even if the information for the CRL and OCSP points were provided properly, these servers may not properly serve and respond to the requests according to a prior study [2]. Thus, we want to test the CRL and OCSP servers that are provided for this corpus of supposedly benign applications in the wild. The question here is that are all these servers always reachable? If not, how often the malfunctioning may happen?

Error	**#Occurrences***	**#URLs****	**Issuer[s]**
Name or service not known	4147	6	AOL, Globaltrust, NLB, Starssl, safescrypt, Swisssign
No certificate subject alternative name matches	2026	1	Verisign
Error 404 not found	2023	1	Swisssign
Read error in headers (connection reset by peer)	2017	2	SSL.com, safescrypt
Error 403 forbidden	62	1	Godaddy
Error 503 backend unavailable connection timeout	19	6	Globalsign
Error 503 first byte timeout	11	6	Globalsign
Error 500 internal server error	6	1	NLB
Error 503 service unavailable	2	1	Swisssign
Read error in headers (connection timeout)	1	1	safescrypt
Error 503 service temporarily unavailable	1	1	Globalsign
Total	10315	27	

*Depicts total number of occurrences for the corresponding problem.
**Depicts number of distinct OCSP URLs that are observed having the corresponding problem.

Table 13: Observed Errors for Unreachable OCSP URLs

We extract 117 distinct CRL points and 48 distinct OCSP points from our set of binaries. We examine the reachability of the CRL and OCSP URLs from December 21, 2019 to July 11, 2020 every two hours. In total, 2418 requests were sent to each CRL and OCSP servers. In Table 12 we report the frequency of each error that we receive for the unreachable CRL URLs. In total, 4.58% (12,976/282,906) of the requests failed. In other words, 4.58% of the code signing certificates are validated without completing the revocation checking due to the soft-fail policy of Authenticode

validation. We categorize the unreachability problems based on the errors. As the total number of occurrences indicates, 61.53% (8/13) of the observed problems are temporary; that is they were not persistent in the course of all the attempts. Yet, if any of these problems occurs upon validation checking of a certificate, the revocation checking would be dismissed.

As indicated in Table 12 and 13, respectively 25.64% (30/117) of the CRL URLs and 56.25% (27/38) of the OCSP URLs were unreachable at least once throughout the time of our test. We categorize the unsuccessful requests to both CRL and OCSP servers based on the error code that we receive. Variety of the observed errors is an evidence of unreliable nature of network. Thus, Authenticode validation cannot rely on the revocation checking using CRL/OCSP servers. The number of occurrences and the URLs that encountered each error imply that these network problems were not commonly persistent. Furthermore, the origins of all these failures were not located at the server side. So, some of these network issues are inevitable and the reachability of the CRL and OCSP servers cannot be guaranteed. On the other hand, since revocation is the critical defense mechanism against notorious abuse scenarios of Authenticode, constant availability of the revocation information is essential. Therefore, the soft-fail policy of Authenticode leaves a significant breach in the validation checking of code signing certificates.

Chapter 5

Discussion and Recommendations

In this section we briefly mention challenges of this study and discuss the insights that we gain throughout our analysis. Furthermore, we provide suggestions to improve the usability and effectiveness of the Authenticode mechanism.

This thesis presents a systematic analysis of Authenticode signed benign applications. We tackle the challenges of studying code signing certificates by designing two frameworks for data collection and analysis. As part of these frameworks, two tools are developed and are in the process of release for the public use of researchers: An application crawler that automates collection of distributed applications on the web and, an Authenticode linter that codifies the baseline requirements and investigate for the potential security issues and bad practices.

In the course of our study, we encounter ambiguous error messages for both of the signing and validation processes of Authenticode. These prompt messages are not specific enough and use technical jargon. Moreover, they do not provide directions for resolving the issue or further action. The lack of thorough documentation for Authenticode would worsen this issue. Not only these issues will impact the usability of this security mechanism, but also it makes the external evaluation and analysis of Authenticode challenging for researchers. For our analysis, we rely on the baseline requirements that are set by CA/Browser Forum Code Signing Working Group in addition to the limited documentation of Authenticode.

Our results suggest that Authenticode does not strictly follow the baseline requirements and its functionality is not always in compliance with its own documentation. Successful validation of padded signatures or violating certificates are examples of the

inconsistency. We identify exceeding validity period, improper dissemination of revocation information (missing CRL/OCSP URLs or unreachable CRL/OCSP servers), usage of weak hash algorithms such as SHA1 and MD5, usage of weak public key algorithm, improper checking of certificate authorization (missing Key Usage and Extended Key Usage extensions or invalid bit settings for Key Usage or Extended Key Usage extensions) and missing certificate policies are violations and bad practices conducted by the certificate authorities or software publishers.

We generate a set of synthesized code signing certificates so that we can do further investigation regarding the violation cases. We leverage OsslSignCode [4], an openssl-based signing tool, in addition to Signtool [8] for signing binaries. Not all the synthesized certificates could satisfy the minimum requirements requested by Signtool for signing. But, OsslSignCode approved all the certificates for signing. We observe that those binaries that are signed using OsslSignCode could be validated by Windows Authenticode. This observation shows that the requirements of the Authenticode validation process are not the same as the requirements of Windows signing process. Therefore, if a software publisher uses a third-party signing tool, there is no guarantee that it maintains all the requirements. This can leave a breach of trust in the Authenticode mechanism.

Recommendations. Eventually, we provide two recommendations as workaround solutions for the discussed issues. Adding a logging system to the code signing PKI can help to overcome the distributed nature of the Authenticode signed applications. Timestamping servers are a good option for providing this service to the public since they are readily equipped for similar data recording. The other option can be public scanning websites such as VirusTotal that collect whitelists and blacklists of certificates and their corresponding signed binaries. These platforms are equipped for efficient data collection and analysis. They can extract required information of the code signing certificates from signed binaries and store them for public use. Certificate authorities can demand a mandatory additional step for their customers to submit their signed applications to these logging systems. This transparency that a logging system can provide to the code signing PKI can mitigate the abuse. Besides, it makes extended test and research on this infrastructure more feasible; considering

that research has significantly contributed to the health and improvement of TLS certificates.

We provide another recommendation for improving the effectiveness of revocation checking. A simple workaround for this problem is providing user with an informative message that explains that validation check could not be completed for a specific reason such as a network problem. This allows a user to make an informed decision regarding the proceeding with the application. The other workaround is blacklisting certificate authorities that do not maintain proper revocation information. Besides, Authenticode can assign reputation to each CA, based on frequency of successfull OCSP/CRL requests.

Bibliography

[1] Omar Alrawi and Aziz Mohaisen. Chains of distrust: Towards understanding certificates used for signing malicious applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 451–456, 2016.

[2] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. Systematic parsing of x. 509: eradicating security issues with a parse tree. *Journal of Computer Security*, 26(6):817–849, 2018.

[3] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. Systematic parsing of x.509: eradicating security issues with a parse tree. *Journal of Computer Security*, 26(6):817–849, 2018.

[4] OpenSSL based signcode utility. https://sourceforge.net/projects/osslsigncode/.

[5] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129. IEEE, 2014.

[6] RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://tools.ietf.org/html/rfc5280section-4.2.1.12.

[7] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. Symcerts: Practical symbolic execution for exposing noncompliance in x.509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 503–520. IEEE, 2017.

[8] SignTool: command-line tool for digitally signing files. https://docs.microsoft.com/en-us/windows/win32/seccrypto/signtool.

[9] Tudor Dumitraş and Darren Shou. Toward a standard benchmark for computer security research: The worldwide intelligence network environment (wine). *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 89–96, 2011.

[10] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the https certificate ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 291–304, 2013.

[11] Nicolas Falliere, LO Murchu, and Eric Chien. W32. stuxnet dossier. symantec. 2013.

[12] Microsoft Security Advisory: Update for deprecation of MD5 hashing algorithm for Microsoft root certificate program. https://support.microsoft.com/en-ca/help/2862973/microsoft-security-advisory-update-for-deprecation-of-md5-hashing-algo.

[13] Baseline Requirements for the Issuance and Management of Publicly-trusted Code Signing Certificates. https://cabforum.org/wp-content/uploads/baseline-requirements-for-the-issuance-and-management-of-code-signing-certificates.v.1.2.pdf.

[14] Microsoft Windows Authenticode Portable Executable Signature Format. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/authenticode_PE.docx.

[15] Virustotal free online virus, malware and url scanner. http://www.virustotal.com/.

[16] WinVerifyTrust function. https://docs.microsoft.com/en-ca/windows/win32/api/wintrust/nf-wintrust-winverifytrust?redirectedfrom=msdn.

[17] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl

certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49, 2012.

[18] Chktrust: Verify if an PE executable has a valid Authenticode. https://linux.die.net/man/1/chktrust.

[19] IBM Security: New Destructive Wiper "ZeroCleare" Targets Energy Sector in the Middle East. https://www.ibm.com/downloads/cas/oaj4vznj.

[20] Microsoft Security Advisory: Changes in Windows Authenticode Signature Verification. https://docs.microsoft.com/en-us/security-updates/securityadvisories/2014/2915720.

[21] SHA-1 is a Shambles. https://sha-mbles.github.io/.

[22] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. Certified malware: Measuring breaches of trust in the windows code-signing pki. *In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1435–1448, 2017.

[23] Doowon Kim, Bum Jun Kwon, Kristián Kozák, Christopher Gates, and Tudor Dumitraș. The broken shield: Measuring revocation effectiveness in the windows code-signing {PKI}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 851–868, 2018.

[24] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified pup: abuse in authenticode code signing. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 465–478, 2015.

[25] Kristián Kozák, Bum Jun Kwon, Doowon Kim, and Tudor Dumitraş. Issued for abuse: Measuring the underground trade in code signing certificate. *arXiv preprint arXiv:1803.02931*, 2018.

[26] Deepak Kumar, Zhengping Wang, Matthew Hyder, Joseph Dickinson, Gabrielle Beck, David Adrian, Joshua Mason, Zakir Durumeric, J Alex Halderman, and Michael Bailey. Tracking certificate misissuance in the wild. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 785–798. IEEE, 2018.

[27] Bum Jun Kwon, Virinchi Srinivas, Amol Deshpande, and Tudor Dumitraş. Catching worms, trojan horses and pups: Unsupervised detection of silent delivery campaigns. *arXiv preprint arXiv:1611.02787*, 2016.

[28] Authenticode Examiner Library. https://github.com/vcsjones/authenticodeexaminer.

[29] Windows Enforcement of SHA1 Certificates. https://social.technet.microsoft.com/wiki/contents/articles/32288.windows-enforcement-of-sha1-certificates.aspx.

[30] Virusshare.com repository. http://virusshare.com/.

[31] Richard Rivera, Platon Kotzias, Avinash Sudhodanan, and Juan Caballero. Costly freeware: a systematic analysis of abuse in download portals. *IET Information Security*, 13(1):27–35, 2019.

[32] R.S.: Flamer: Highly Sophisticated Security and Discreet Threat Targets the Middle East (2012). http://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east.

[33] Microsoft Defender SmartScreen. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-smartscreen/microsoft-defender-smartscreen-overview.

[34] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Md5 considered harmful today, creating a rogue ca certificate. In *25th Annual Chaos Communication Congress*, number CONF, 2008.

[35] Sigcheck Windows Sysinternals. https://docs.microsoft.com/en-us/sysinternals/downloads/sigcheck.

[36] Cong Tian, Chu Chen, Zhenhua Duan, and Liang Zhao. Differential testing of certificate validation in ssl/tls implementations: an rfc-guided approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–37, 2019.

[37] James Walden. Censys: a better ipv4 scan search engine. 2016.

[38] Mike Wood. Want my autograph? the use and abuse of digital signatures by malware. In *Virus Bulletin Conference September*, pages 1–8, 2010.

[39] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

Appendix A

Authenticode Verification Error Messages

Problem	Error Message	Error Code
Bad Digest	The digital signature of the object did not verify	0x80096010
Malformed Signature	The digital signature of the object is malformed. For technical detail, see security bulletin MS13-098	0x80096011
Expired	A required certificate is not within its validity period when verifying against the current system clock or the timestamp in the signed file	0x800B0101
Revoked	A certificate was explicitly revoked by its issuer	0x800B010C
Untrusted Root	A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider	-
Crypto Policy	Signature did not pass crypto policy	-
Chaining	A certificate chain could not be built to a trusted root authority,	0x800B010A
Test Root	The certification path terminates with the test root which is not trusted with the current policy settings	0x800B010D

Table 14: Authenticode Verification Errors

Expired Certificate Error Message

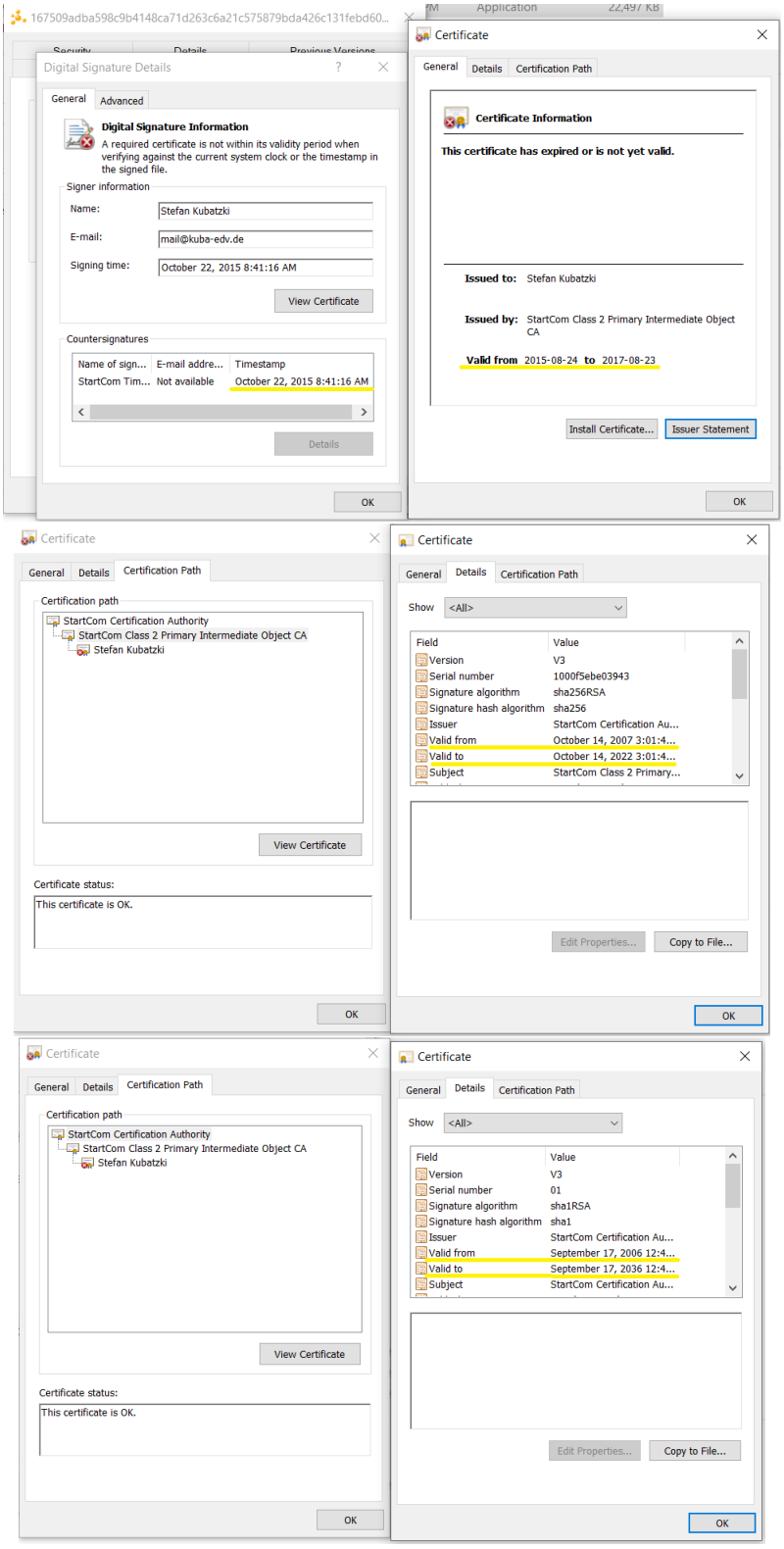

Figure 4: Expired Certificate