CHAPTER
14

Detection and Prevention

Application developers use various mechanisms to detect and protect against reversing and cracking. However, some of these methods are more effective than others. This chapter explores some of the most common techniques, their relative strengths and weaknesses, and how they can be defeated.

CRC

A cyclic redundancy check (CRC) is a mathematical calculation performed on the bytes of the data to be protected. The result is stored as the CRC, which is often appended to the data (i.e., `data data data data data data CRC`). To verify the data, recalculate and compare.

CRC algorithms have their advantages, including the following:

- Fast and compact
- Easy to accelerate with hardware
- Quick to calculate and compare
- Numerous options available (IEEE802.3, CRC-32, etc.)

In general, CRCs are great for detecting accidental errors or modification, such as transmission errors.

© 2024 John Wiley & Sons, Inc. Published 2024 by John Wiley & Sons, Inc.

However, they are a poor defense against intentional errors or modifications. CRCs can be easily recalculated and updated by an adversary. For example, a simplistic CRC might add all of the bytes together and save the result. If a corruption were to occur in the file somewhere in the data, then the new sum would not match, and action could be taken. If the corruption occurred in the CRC portion of the file, then the sum would not match the corrupted CRC, and action could be taken. This is great for detecting if a bit got accidentally flipped while being downloaded, for example.

But because the CRC is so trivial to recalculate, it's simple for an attacker to make their modifications and simply update the CRC to include their new values.

Is This a Strong Protection?

Comparing CRCs to the CIA triad yields disappointing results:

- **Confidentiality:** None
- **Integrity:** Very little (it's too easy for an attacker to recalculate and put the new CRC into the file)
- **Availability:** None

This defense can easily be defeated by generating a new, valid CRC. Alternatively, you can simply patch out the CRC check. CRCs are powerful for detecting accidental corruption but are not useful for intentional corruption.

Code Signing

Many organizations digitally sign their code before releasing it. This is because code signing provides two main benefits:

- **Authenticity:** A digital signature can be generated only with the correct private key. This proves that software came from its alleged creator.
- **Integrity:** Changing digitally signed data invalidates its signature. Code signing proves that software hasn't been modified after release.

Code signing protects against a wide range of potential attacks. However, from a cracker's perspective, the most significant impact is that it can prevent patching if a program checks its signature before executing.

How to Code Sign

Code signing works using public key or asymmetric cryptography. These cryptographic algorithms use a pair of public and private keys. To code sign, you first need to generate a public/private keypair.

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Digital signatures are validated using your public key; however, you need a way to prove that a particular public key belongs to you. This is where public key infrastructure (PKI) comes into the picture. Using the generated public key, you apply for a certificate from a code signing certificate authority (CA). The CA will verify your identity and issue a digital certificate, which contains your public key and validates your ownership of this.

With this certificate, you can now generate digital signatures. To do so, you would generate a hash of the executable and encrypt that hash with the private key. Then, when you distribute the executable, you would bundle the resulting signature and your digital certificate with the executable.

While you can go through this process manually, many build tools will do this for you. You still would need to buy a certificate and load it into your build tool, but then you can ask the build tool to sign the application. If this is your first exposure to PKI, know that this is intentionally just scratching the surface of it; there are many books dedicated to just this concept.

How to Verify a Signed Application

A code signature is essentially an encrypted hash of the executable. After verifying that the public key is valid using the associated certificate, you can decrypt the executable's hash. Then, you independently calculate the hash of the application using the same hash function as the application developer. If you compare the two hashes and they match, the application is authentic and unmodified. If they differ, the application is fake or has been tampered with.

Most operating systems will verify code signatures for you. The OS will also generate a warning if the public key used to generate the code signature is unverified, as shown in Figure 14.1. However, most people will click Run anyway.

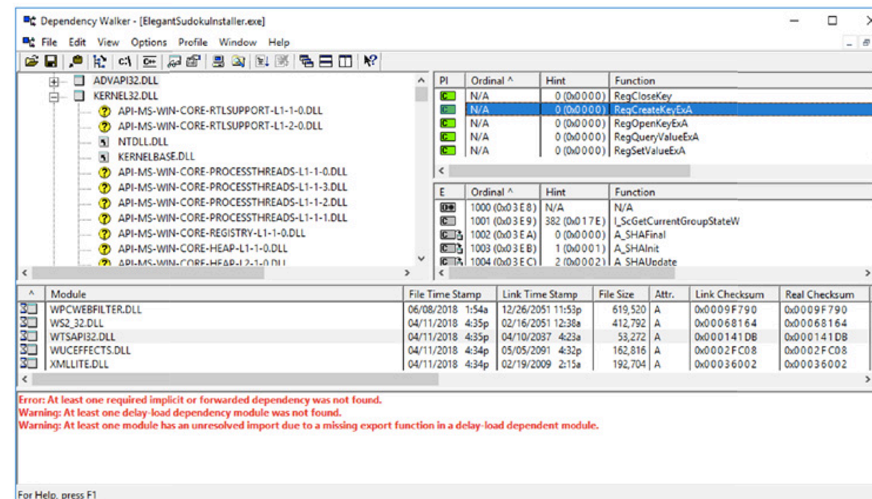

Figure 14.1: Windows warning of unverified program

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Is Code Signing Effective?

Does code signing stop all patching attacks? No.

The reason for this is that there must be a piece of unsigned code that does the sign checking. This includes performing a few actions:

1. Calculate the hash of the code.
2. Check if this is what it should be.
   2a. If the answer is correct, run code.
   2b. If the answer is incorrect, don't run code.

This signature verification code can't be included in a code signature because it needs to contain (or access) the hash value to compare against. It's impossible to predict what this value would be without hashing the application. If you hashed the application (which includes this value) and included the hash in the application, then the modified application would have a new hash.

Since the signature verification code can't be signed, there is a different location that could be patched to bypass code signing. However, code signing is hands down one of the best techniques for securing software integrity against both accidental and intentional modifications.

Code Signing vs. CRC

CRCs are commonly used to detect bit errors in data sent over a network. However, they provide protection only against accidental changes, not intentional ones. CRCs are easily recalculated by an adversary.

Code signing is as strong as your protection of the private key. Without the private key, an attacker cannot regenerate a valid signed hash.

Is This a Strong Protection?

Code signing does a lot better than CRCs when compared to the CIA triad.

- **Confidentiality:** None
- **Integrity:** Yes! Fantastic
- **Availability:** None

A more difficult approach to defeating code signing is to steal the private signing keys and use them to digitally sign a modified version of the application.

RASP

Runtime application self-protection (RASP) embeds security into the running application. It does so by intercepting system calls and verifying that they are

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

from an expected source. It also intercepts data manipulations and verifies that they are coming from authorized sources.

RASP is a reactionary defense. It can be configured to "stop" attacks live. For example, RASP can do the following:

- Drop/delete a call it deems malicious, such as a suspicious SQL call into an application.
- End a user session.
- Halt execution.

Function Hooking

One technique that RASP uses is function hooking. This involves overwriting the first few bytes of a function's code with a jump to the RASP code.

The RASP code will include checks to verify that the call is legitimate. This can include the following:

- Checking the parameters and context of the call
- Checking the code has not been modified (might compare a hash of the function with a known good hash)

At the end of the RASP code, it will then execute the overwritten code before jumping back to the original function.

Risks of RASP

If RASP detects an attack, it can stop execution. However, this may not be acceptable depending on the use case of your software. For example, in hospitals, manufacturing, critical infrastructure, automobiles, and similar environments, an application suddenly halting can pose a significant risk to health and safety.

RASP can also have its downsides even in the absence of an attack. Some effects include the following:

- **Speed:** Because of the function hooking, RASP has a nontrivial effect on speed.
- **Size:** Function hooking and lookup tables help to assure security; however, they also bloat binaries.

Is This a Strong Protection?

RASP provides mixed results when compared against the CIA benchmark:

- **Confidentiality:** No
- **Integrity:** Yes (for the sections that RASP is protecting, the context checking at runtime is a very powerful check)
- **Availability:** No (in fact can be negative)

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

If RASP is correctly configured, it can be difficult to defeat. You can't easily patch the application if code signing is enabled, and you can't easily reverse the application if anti-debugging is enabled.

However, RASP does open a potential avenue for an attack against availability. If you can identify an input that it perceives as an "attack," you can potentially make it shut itself down.

Allowlisting

Allowlisting, sometimes called whitelisting, is providing the execution environment with a list of "good" things. For example, a computer may allow only allowlisted applications to run.

There are numerous commercial products that provide allowlists. For example, the Windows operating system has built-in *software restriction policies*.

From a cracking perspective, allowlisting can prevent the use of cracking and reverse engineering tools. For example, tools such as Procmon, debuggers, Cheat Engine, ResourceHacker, Dependency Walker, and other reversing and cracking applications are unlikely to be included in the allowlist.

Allowlisting is difficult to get right. It can be difficult to know all of the various libraries that your application needs. When generating a whitelist, a great deal of testing must be performed to ensure that all required applications and libraries are included on the allowlist.

How Allowlisting Works

There are two main approaches to allowlisting. A list can be based on a process's name or on its hash. These lists are applied only when an application is first launched.

Breaking Name-Based Allowlists

Allowlists keep track of the names of processes or applications allowed to execute. To bypass this type of list, name your malicious application a whitelist-approved name. For example, you determine that `solitare.exe` is allowlisted, so you name your malicious app `solitare.exe`.

Breaking Name and Hash-Based Allowlists

If an allowlist uses both application names and hashes, it can't be bypassed by renaming an application. The hash of the malicious app wouldn't match that of the legitimate one.

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

However, these allowlists can be defeated through process injection. Once an allowlisted application is running, if you can get code execution, you can inject malicious libraries. And while that's easily said, getting code execution is not often trivial. So, this is one of those cases where it sounds easy because it can be said in a sentence, but in reality having a prerequisite of code execution in a whitelisted process may be a full roadblock for a cracker.

If you have gotten the elusive code execution in an allowlisted application, there are numerous techniques for loading into the process. In Windows, you can use `LoadLibrary()` or `SetWindowsHookEx()`. In Linux, you can use `ptrace()`/`PTRACE_POKEDATA`/opcodes for `uselib()` syscall.

An application's hash is checked prior to application launch. Modifications to the application after it is launched won't be detected by the allowlist.

Example: Metasploit

Metasploit is a popular hacking tool. Its main goal is to exploit an application and inject a meterpreter, which provides the attacker with remote access to the infected computer. (See the "Tools" section of our repository for links.)

With Metasploit, no new applications are started; a meterpreter injects into the hacked process. From there it can "pivot" into any other running application.

Is This a Strong Protection?

Allowlisting provides limited protection:

- **Confidentiality:** No
- **Integrity:** Yes (if paired with the name and hash; however, the integrity checking is generally done only at application start time)
- **Availability:** No

Allowlisting can be defeated in a couple of ways. A malicious program can impersonate a legitimate application to defeat a name-based whitelist or use code injection to defeat an allowlist that uses both names and hashes.

Blocklisting

Blocklisting, sometimes referred to as blacklisting, is the exact opposite of allowlisting. Instead of specifying everything that is permitted, it is a list of all the things that are not allowed. The blocklist can be based on names, keys, or hashes.

Blocklists are easy to make but difficult to maintain. For example, consider a blocklist including the malicious executable `virus1.exe`. What happens next week when `virus2.exe` comes out?

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

From a more cracking perspective, you might blocklist keys that you know to be bad (i.e., cracked). Depending how your key generation works, it may be possible to blocklist whole subsets of keys.

Alternatively, a program can also refuse to run if certain other applications are seen. For example, the application may not run if a debugger is installed.

Many antiviruses use this approach to identify and block known malware. They include a list of "signatures" of known bad applications. If something matches the signature, it's flagged as bad.

Is This a Strong Protection?

A blocklist provides less protection than a whitelist:

- **Confidentiality:** None
- **Integrity:** Some (if paired with hashes or keys)
- **Availability:** None

The means of defeating a blocklist depend on the information that it uses to identify malicious applications. If it is name-based, change the name. If it stores the hashes of known-bad programs, mutate it by making a small change to the application's code or data to change its hash.

Remote Authentication

For most anti-reversing and anti-cracking strategies, the attacker has all of the pieces that they need to overcome the defense. With enough time, they can reverse engineer and/or patch the application.

Remote authentication requires the application to retrieve something remotely in order to work. For example, it might get a key from a remote server that it uses to decrypt some crucial code.

Most attackers will reverse engineer a system "offline." They don't want it reaching out to your servers because they don't want you to have their IP address or to know that they are running your software. Keep in mind when attempting to crack a piece of software, you're likely launching and running the startup and checking code frequently. Whereas a legitimate user would likely launch the application at max a few times a day. That type of behavior is really easy to spot on a remote authentication server. A user who is authenticating 100 times a day is likely doing something nefarious.

Architecting the application in such a way that it can't run without information from a server helps prevent reverse engineering. The attacker will either need to reverse it "online" or give up.

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Remote Authentication Example

One possible approach to implement remote authentication is to encrypt every part of the application except for the loaders. The loader sends system information, a hash of the software, and the activation key to the server.

The server will verify the expected hash and the activation key. If they validate, it uses an algorithm to produce a decryption key, which it will send back to the application. The loader can then decrypt the application, enabling it to run.

An attacker will not be able to "mimic" your remote server and algorithm without access to the server-side code. The only way to research the software will be to activate it online. The application can have decryption code be resident in memory only. This way, each startup requires server interaction.

The main challenge of this approach is that implementing cryptography and enterprise key management solutions is not trivial. A mistake may allow a cracker to bypass the validation code and generate their own decryption keys. As discussed with packed applications, once it's unpacked in memory, you could take a memory dump of it for future static analysis. However, that memory dump can't easily (or sometimes not at all) be turned into a decrypted application capable of running. The memory dump won't be useful for patching or testing modifications but never discount the value of static analysis.

Is This a Strong Protection?

Remote authentication provides mixed reviews when compared against the CIA triad:

- **Confidentiality:** Some (the application will eventually be decrypted in memory, but the binary at rest is restricted)
- **Integrity:** Yes (the server should be doing some type of integrity checking prior to releasing the response)
- **Availability:** Possibly negative

One possible attack against a remote server is to set up a fake server. To start, activate the application online and capture all communication between the application and server.

Then, stand up a fake server with appropriate responses. The application's code will be decrypted and can be saved to disk.

This approach does require a single online application to get the decrypted code. However, this allows a full, decrypted binary to be created, making further online authentication unnecessary. But note that this approach can't always work if the application does good due diligence in requiring certain certificates from the server or if the challenge/response from the server is not always the same (i.e., changes with time or date).

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Lab: ProcMon

This lab shows that there is more than one possible way to crack a program. Head back to the book's GitHub page (`https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES`) and locate the ProcMon lab.Skills.

This lab uses ProcMon and IDA to understand opportunities for alternative cracking solutions. Some key skills being tested include the following:

- Analyzing program behavior dynamically
- Identifying indirect approaches to circumventing software defenses

Takeaways

Watching what a process does from the outside can be quicker/easier than watching it from the inside (that is, debugging is not always the best approach).

There are usually many ways to crack a program; finding the best takes practice.

Summary

This chapter presented various methods of protecting against software cracking and reversing. Some techniques are generally ineffective, while others can work but also have some downsides.

It's important to remember that almost any defense can be defeated given enough time and effort. The goal is to slow an attacker down and, ideally, make them frustrated enough to give up.

10.1002/9781394277131.ch14, Downloaded from https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/doi/10.1002/9781394277131.ch14. By University Of Ontario, Wiley Online Library on [23/04/2026]. See the Terms and Conditions (https://onlinelibrary-wiley-com.uproxy.library.dc-uoit.ca/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License