Protecting Better with Themida, WinLicense and Code Virtualizer

Themida

Advanced Windows software protection

by Oreans Technologies

Themida is an advanced Windows software protection system, developed for software developers who wish to protect their applications against advanced reverse engineering and software cracking.

WinLicense

Advanced Windows software protection & licensing

by Oreans Technologies

WinLicense combines the power of software protection (as Themida), with the power of advanced license control. It offers a wide range of powerful and flexible techniques that allow developers to securely distribute trial versions of their applications.

Code Virtualizer

Total Obfuscation against Reverse Engineering

by Oreans Technologies

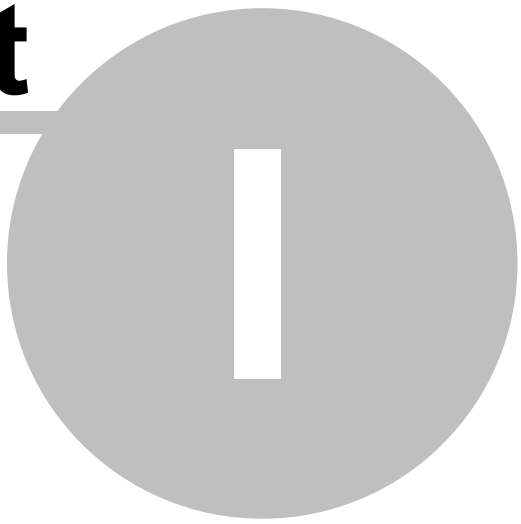
Code Virtualizer is a powerful obfuscation system for Windows, Linux and Mac OS X applications that helps developers to protect their sensitive code areas against Reverse Engineering with very strong obfuscation code, based on code virtualization.

Protecting Better with Themida, WinLicense and Code Virtualizer

Printed: November 2024

Table of Contents

Part



1 Introduction

The goal of this guide is to assist developers in enhancing the security of their software using the SecureEngine Technology (which is available in Themida, WinLicense and Code Virtualizer). This guide is crafted for everyday programmers who may not be familiar with software security, providing them with basic yet effective strategies to bolster the security of their applications.

Utilizing the SecureEngine Technology can be an excellent strategy to safeguard your software when applied correctly within your application. However, if a developer haphazardly inserts the SecureEngine Technology protection macros into their application without adequately shielding crucial routines, it could be easy for crackers to bypass the protection, almost as if the application wasn't protected at all.

It is crucial to identify and understand which routines might be targeted by crackers within your application. Focusing your protective efforts on these vulnerable routines with the SecureEngine Technology is key. It is unnecessary to secure functions that hold no value for a cracker, such as functions that operate identically in both trial and registered modes.

We aim to regularly update this guide with new information and insights to help you enhance the security of your applications in collaboration with Themida, WinLicense or Code Virtualizer. Moreover, we encourage you to implement your own ideas to fortify your application's security. If you wish to share your strategies with us for potential inclusion in this guide, we warmly welcome your contributions!

Part

2 Test Cases

In this section, we will explore how to utilize the SecureEngine protection macros in various scenarios. You don't have to implement all the suggested solutions in your application; just choose the one(s) that best suit your specific needs.

The key is to pinpoint the sensitive code areas within your application - these are parts of the code that are crucial for a cracker to scrutinize in order to understand how your application functions, potentially allowing them to alter or redirect the execution flow. It's vital to shield these sensitive code areas using the SecureEngine protection macros. In the upcoming topics, you will find examples illustrating different types of sensitive code areas, assisting you in identifying similar zones within your own application.

2.1 Trial/Registration Scheme

Many applications operate under a "Trial/Registration" scheme. This means the same application can function in either trial or registered mode, depending on whether it has been activated with a valid license or serial number. The method you use to verify if your application is registered can vary. In the sections that follow, we will present different schemes that can be adapted for various applications.

2.1.1 Global Variable

The scheme described below relies on a global variable to determine if the application is operating in trial or registered mode.

Usually, the application will feature a routine that checks and validates a license, storing the result in a global variable (we'll refer to it as "`is_registered`") that indicates whether the application has been registered or not. Throughout various routines in the application, the code references the `is_registered` variable to dictate different paths of execution based on the registration status. In this scheme, the "`is_registered`" variable is a critical point of focus.

Here, there are two primary sensitive code areas to be aware of:

1. The routine that inspects and verifies the validity of the license.

2. Every instance in your code where the `is_registered` variable is accessed or modified.

It's essential to secure all access points to the `is_registered` variable. Failure to do so can allow a cracker to locate various checkpoints where the variable is assessed to control important code pathways (pertaining to trial or registration). This could expose a vulnerability in your trial/registration scheme, making it easier for them to breach your application. They would simply need to modify the in-memory value of the `is_registered` variable, bypassing the protective layers offered by the SecureEngine technology.

Example (In red you can see all sensitive code areas):

```
int main(void)
{
```

```
        VIRTUALIZER_START

        // some code

        is_registered = CheckLicense();

        // some code

        VIRTUALIZER_END
}

int MyFunction1(void)
{
        VIRTUALIZER_START

        if (is_registered)
        {
                // execute code for registered version
        }
        else
        {
                // execute trial code
        }

        VIRTUALIZER_END
}
```

In the example mentioned above, it's evident that all the sensitive code sections are encompassed between the VIRTUALIZER_START/END markers. Make sure not to leave any sensitive code area without a VIRTUALIZER macro!

Tips to protect better

- To enhance security, consider not limiting the **`is_registered`** variable to a boolean value. Instead, assign a random number to the **`is_registered`** variable to prevent it from serving as a conspicuous flag (zero or one) that might attract the attention of a cracker. If it's feasible, populate the **`is_registered`** variable with a different value each time the application runs – this approach would offer even greater protection.

- Furthermore, if your "`CheckLicense()`" function primarily exists to assign a value to a global variable, it might draw the attention of a cracker. This is particularly likely if the function is fully virtualized and solely assigns a value to a global variable upon exit, clearly indicating to the cracker that the "global variable" holds significant meaning. To mitigate this, don't limit the function to just verifying the license. Incorporate additional initialization tasks or functionalities within this function. This way, various variables will be engaged and altered, and a diverse range of code (beyond just license verification) will be executed, thereby diluting the focus on the license check.

2.1.2 Registration Function

The scheme outlined below revolves around a function that is invoked each time the application needs to determine if it's operating in the registered mode. Unlike the previous scheme, this one doesn't rely on a global variable to keep track of whether the application is in trial or registered mode.

In this scenario, there are two sensitive code areas to pay attention to:

1. The routine that verifies the license's validity, which we will refer to as the "`CheckLicense()`" function.
2. Every instance where the "`CheckLicense()`" function is called in your code.

It is imperative to secure all the routines where the `CheckLicense()` function is invoked. If not, a cracker might identify that the virtualized function (referred to here as "XXXXX", which correlates with `CheckLicense()`) triggers different execution paths depending on whether it's operating in trial or registered mode. Consequently, they could simply alter the entry point of the `CheckLicense()` function to return the "registered" status, bypassing the virtualized code entirely. This would enable them to breach your application without having to navigate through the protective virtualized code barriers.

Example (In red you can see all sensitive code areas):

```
bool CheckLicense(void)
{
	VIRTUALIZER_START

	// your code that reads and check the license

	// return if license is valid or not

	VIRTUALIZER_END
}

int MyFunction1(void)
{
	VIRTUALIZER_START

	if (CheckLicense())
	{
		// execute code for registered version
	}
	else
	{
		// execute trial code
	}

	VIRTUALIZER_END
}

int MyFunction2(void)
{
	VIRTUALIZER_START

	if (CheckLicense())
	{
		// execute code for registered version
	}
	else
	{
		// execute trial code
	}

	VIRTUALIZER_END
}
```

In the example provided above, you can observe that all sensitive code regions are enclosed within the VIRTUALIZER_START/END markers. Ensure no sensitive code area is left without a VIRTUALIZER macro!

Tips to protect better

- To boost your defense mechanisms, consider not relying solely on the `CheckLicense()` function to determine the registration status of the application. Introducing secondary functions (or more) to validate the license could be a stronger strategy. This way, a cracker would be confronted with multiple barriers, having to bypass several functions (like `CheckLicenseType1()`, `CheckLicenseType2()`, etc.) that are tasked with license verification. While a cracker might manipulate the execution pathway of `CheckLicenseType1()`, the conflicting reports from different functions (with one indicating "registered" and another "not registered") could alert you to potential breaches, thus enhancing the security of your application.

2.1.3 Using an External DLL

When you use an external DLL to export a function that determines whether your application is registered, you need to take special care. Crackers can bypass your registration scheme simply by altering the return value of the exported function in your DLL. A common mistake that eases the job for crackers is assigning recognizable names to the exported validation function, like "IsRegistered" or "CheckRegistration". This allows crackers to quickly identify and alter the return value of the function in the DLL, possibly unlocking the "registered" mode of the application.

Here are some straightforward tips:

- **Choose Unrecognizable Names:** Avoid giving your validation function in the DLL an easily recognizable name. Crackers often start by identifying all exported names in an external DLL linked to your application, gaining insights into the DLL's functionality and potentially locating a sensitive registration-related function. Using complex, mangled names for exported functions in a sensitive DLL can be more secure.

- **Implement Multiple Validation Steps:** Don't rely solely on a single exported function in the DLL to determine the registration status. Consider the potential outcomes if a cracker alters the output value of a registration function. Could this enable them to use your application as if it were registered without any issues, or would it cause a crash later on? Despite using a robust validation function with strong cryptography, a cracker might still manipulate the output value following your complex mathematical calculations. To counter this, alongside your primary registration function ("RegFunction1"), implement additional functions that conduct full or partial license validations independently of "RegFunction1". This strategy makes it difficult for crackers, even if they patch "RegFunction1" to return "registered" consistently, as other validation functions may not confirm the "registered" status.

- **Utilize the VIRTUALIZER Macro:** Incorporate a VIRTUALIZER macro within the validation functions of your sensitive DLL. Extend this precaution to your main EXE or other DLLs, especially in sections of the code that call upon the validation function in your sensitive DLL.

- **Enhance DLL Security:** Ensure that your DLL isn't solely responsible for determining the application's registered status. Be aware that some crackers might replace your DLL entirely with a version that bypasses the implemented code, only returning the expected values for all exported functions. This substitution means that your application might end up calling the cracker's DLL instead of your original one. To prevent this, link your DLL closely with your EXE, allowing the EXE to verify the successful loading and operation of your DLL. Concurrently, enable your DLL to confirm that it operates within your EXE's process space, preventing unauthorized use in other applications.

2.1.4 Trial Expirations

Most shareware applications are designed to expire after a set number of days, uses, or on a predetermined date. Generally, the application needs to save this "expiration counter" in a system location that isn't removed when the application is closed or uninstalled. On Windows systems, developers often save the trial expiration information in the Windows Registry or an external file. This data is then checked each time the application starts to read the current expiration status.

Before settling on a storage location for the trial expiration data, consider the following potential scenarios and how your application would respond:

- *What would happen if a cracker adjusts the system clock to a date years before or after the current date and then launches your application for the first time?*

- *How would your application react if the cracker reverses the date by days or years after your application has been launched for the first time?*

- *What would occur if the cracker locates one of the places where the trial information is stored and deletes that Registry entry or specific file?*

To counter these basic attack strategies that crackers (or even ordinary users) might use on an application with trial expiration, it's crucial to develop solutions. A primary step is to not depend on a single storage location for the trial expiration information. Additionally, maintain a record of the current date in a separate location to perform initial checks that confirm the system clock hasn't been turned back since the last application launch. For more strategies on storing trial expiration information, refer to this guide.

It's important to note that the SecureEngine technology cannot secure the information stored in the Windows Registry or external files. However, you should protect all code that accesses sensitive locations (Registry/Files) to read and record the trial expiration data. Ensure that this code is enclosed within a VIRTUALIZER macro.

2.2 Protecting Important Routines

In certain situations, you may possess significant algorithms that you wish to shield from inspection, preventing reverse engineering by competitors. Utilizing the SecureEngine protection macros, you can secure these sensitive routines. However, be aware that the execution of protected (or virtualized) code will operate at a considerably slower pace compared to its original version. Specifically, if your algorithm contains a tight loop (a segment that iterates numerous times), you might want to reconsider safeguarding that specific segment or employ a more lightweight Virtual Machine for its protection.

Solution 1 (Removing the Tight Loop):

```
int MySecretFunction(void)
{
        TIGER_WHITE_START

        // some code

        TIGER_WHITE_END

       // tight loop here. It's outside of the macro markers:

       for (int i = 0; i < 0x100000; i++)
        // your code

        TIGER_WHITE_START

        // some code

        TIGER_WHITE_END
}
```

Solution 2 (Employing a Lightweight Virtual Machine (e.g., "FISH_LITE") for Your Tight Loop):

```
int MySecretFunction(void)
{
        TIGER_WHITE_START

        // some code

        TIGER_WHITE_END

        FISH_LITE_START

       // tight loop here. Virtualized with FISH_LITE VM:

        for (int i = 0; i < 0x100000; i++)
        // your code

        FISH_LITE_END

        TIGER_WHITE_START

        // some code

        TIGER_WHITE_END
}
```

2.3 Scheme for Serial Numbers Registration

Applications often require registration via a serial number. Here, we share insights on fortifying your serial number scheme against potential cracking attempts:

- **Embed the Verification Routine in a "VIRTUALIZER" Macro:** Ensure the routine that validates the serial number is housed within a "VIRTUALIZER" macro. This step forms a primary line of defense against tampering.
- **Implement Multi-Faceted Initialization Steps Within the Validation Routine:** Crackers may attempt to manipulate the return value of your validation routine. To counter this, avoid over-reliance on the return value alone. Instead, initiate other steps within the validation routine to prepare your application to operate in registered mode.
- **Use Serial Numbers to Influence Code Execution Flow:** When feasible, utilize digits from the input serial number to dictate varied paths in your code's execution. For instance, configure your serial number such that the third number (s3) equates to the sum of the first (s1) and second (s2) numbers. This logic can then be integrated into a separate function that is activated only in registered mode:

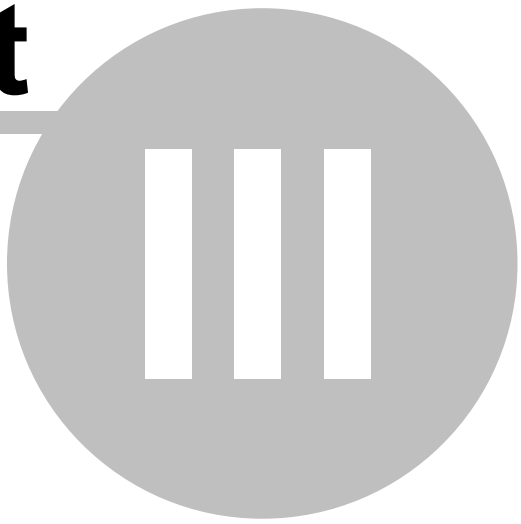
```
// if the validation routine was patched, a different function will
// be called here, hence it will crash or produce a different functionality

ExecuteFunction(TableFunctions + XXXX + s3 - s2 - s1);
```

Adopt Late Crash Protocols: Should you detect a compromise in the validation routine, consider implementing a "late crash" strategy or altering the application's behavior. A "late crash" strategy avoids immediate error notifications like "Cracking attempt detected." Instead, retain a note of the breach (possibly in a variable) to indicate your application is undergoing patching. At a later point, you can induce a crash in a different function or delay the response to the detected attack by several minutes.

This guide outlines a basic framework for bolstering the security of your serial number scheme. By incorporating additional checks, you can significantly amplify the protective measures in your application.

Part



3 General Protection Advise

This section outlines some universal protection strategies that can significantly hinder crackers' attempts to breach your application.

3.1 Delayed Cracking Detection

In many cases, you might notice that your application is being hacked or has been altered partially. For instance, you can set up two different processes/functions that verify a license or a serial number, and both should give the same result if a valid license or serial number is entered.

Usually, a hacker might notice the first process/function that checks the license or serial number and will attempt to bypass or modify it to get the desired outcome. Your second verification process/function will then return the actual result, helping you realize that the first process/function has been compromised. At this point, avoid displaying error messages like "Incorrect License" or "Cracking Attempt Detected," as the hacker will quickly understand that there's another process performing additional checks and will attempt to find and modify that as well.

Instead of alerting the hacker that something has gone awry after the patching is detected, save this information in an internal variable or flag indicating that your application is in the "cracking stage." Later, when you observe that your application is in the "cracking stage," alter its behavior or generate an error when a specific function is performed. This makes it harder for the hacker to identify the issue's source. For instance, if your application stores data in a database, a file, or on a CD/DVD, you could start copying incorrect information to the database/file/DVD when you detect that your application is in the "cracking stage," causing it to operate incorrectly but continue running.

Another option is to wait a few hours (or even days!) once you realize that your application is in the "cracking stage" before causing it to crash or malfunction. Of course, this approach is not suitable on some applications.

3.2 Protecting Sensitive Strings

Sensitive strings are specific phrases or words that could potentially aid crackers in attacking vital validation routines or in realizing that their cracking attempt was unsuccessful. This can occur, for instance, when a message such as "cracking/patching detected" is displayed.

Typically, all strings are visible within an application when it is running, even if you are using a software protector. Despite protection measures, these strings or data have to be decrypted when the application is operating so it can function as expected. Hence, it is prudent to keep sensitive strings encrypted even during the compilation phase. This way, you only decrypt them when they are about to be displayed during runtime.

SecureEngine facilitates this process through the "**Virtualize Strings**" option found in the Extra Options panel. When you use this option, all the strings mentioned within a VIRTUALIZER

macro will be encrypted during the protection stage and will only be decrypted (automatically) when they are about to be used. Ensure that you encase all code referencing a sensitive string within a VIRTUALIZER_START/END macro. This guarantees their encryption as intended. Here's an example:

```
int MyFunction1(void)
{
        VIRTUALIZER_START

        if (is_registered)
        {
                printf("Your application has been registered");
        }

        VIRTUALIZER_END
}
```

Remember, it's not necessary to encrypt or safeguard all the strings in your application since many of them might not be of interest to a cracker. Generally, strings that are consistently visible in your application (whether in trial or registered mode) are not deemed sensitive.

3.3 Reporting Application Status

It's important to effectively communicate the registration status of your application to your customers. A good way to inform them that the application is operating in "unregistered" mode is by displaying a message in the "About" dialog, which should be generated dynamically if possible. Moreover, the function that verifies whether your application is running in unregistered mode should be separate from the one validating the license.

Avoid relying on a single validation function to verify a license or serial number and determine if your application is in trial or registered mode. Instead, utilize multiple validation functions to ascertain if any of them have been altered or "patched", potentially leading to a delayed crash.

3.4 Slow Validation

In your application, you might have one or more validation routines that check the correctness of entered licenses or serial numbers. Usually, developers create a routine that returns a value indicating whether the validation was successful. However, this makes it easier for crackers to inspect and modify the exit code of the function to see the outcomes of different return values.

Often, developers store the validation outcome in a global variable or another location, quickly displaying a message about whether the validation was successful.

To enhance the security of your main validation routine, try not to instantly notify the user about the validation result. Instead, require a restart of the application to complete the registration. During the initialization phase, check the result of the validation. This strategy will divert the cracker's full attention away from the main validation routine, encouraging them to explore different areas of your program. This also slows down the cracking process, as the cracker will need to restart your application multiple times.

3.5 Utilizing Limited Functionality

If your application has the option to run in either trial or registered mode, it's more beneficial to release two distinct versions of your application. One version should only operate in "trial mode," lacking the "special functionality" that is exclusively available in the "registered mode." Only your customers should have access to this registered version, keeping it from being publicly available.

While it's still possible for the registered version to be distributed without authorization, this strategy essentially grants you more control over who can access the registered version of your application.

However, this approach might not be suitable for every application. In cases where it's not feasible to distribute both a limited and a registered version, we recommend implementing one or more of the strategies outlined in this guide to safeguard your application.

3.6 Relocate Sensitive Data

When validating a license or serial number entered through a registration dialog, it's more secure to transfer and transform this information across different memory areas. Here's how you can do this:

1. First, retrieve the entered serial number.
2. Conduct initial checks and transfer the first transformed version of the serial number to a different memory location.
3. Perform another transformation check and transfer it to yet another memory location, and so on.

Ensure that all these processes are shielded with a VIRTUALIZER macro. This step adds a layer of complexity for potential crackers, making it easier for them to lose track of the serial number as it moves through various memory locations.

3.7 Monitor your Memory Content

A common error some developers make is retaining valid serial numbers within their application or revealing a correct serial number in memory during comparison with the entered one. It is essential to ensure that, while your application remains unregistered or when a user inputs an incorrect serial or registration number, no correct serial or registration numbers are stored in the memory during your computations.

Crackers will scrutinize not only your validation code but also the allocated memory during various phases of the validation process (or at the end of it) to see if any data "magically" manifests in memory. To enhance security, it's advisable to use a CRC (Cyclic Redundancy Check) to verify the accuracy of the entered serial or registration number, instead of juxtaposing the input data with a pre-coded valid serial number.

While it is crucial to encapsulate your validation routine within a VIRTUALIZER macro, it's important to note that tools like Themida, WinLicense or Code Virtualizer secure the instructions in your routine but not the memory data utilized within your code. The memory accessed or modified during your validation process remains unprotected, both in your original and safeguarded application.

3.8 Securing Sensitive Data in the Registry

Applications monitoring trial or registration expiration often store sensitive details within the Windows Registry to track the current expiration status or counter. It's crucial to note that despite using a Software Protector, this information remains visible in the Windows Registry, irrespective of whether the code that writes or accesses the Registry is virtualized or obfuscated.

Take, for instance, an application that records the trial status in the Registry key HKLM/Software/Microsoft/Shared/trial_status. Once a cracker identifies that this Registry key is the storage point for the trial status, they can release a crack that merely deletes that Registry key, resetting the trial on any computer.

A more secure approach would be to create an array of strings (preferably encrypted), termed "ArrayTrialEntries", that lists various Registry key names where the trial status might be stored. Let's assume you have 20 different potential storage locations. During runtime, you extract a "unique" identifier from the current machine (utilizing the CPUID instruction, HDD serial number, BIOS serial, etc.) and perform mathematical operations to generate a number between 1 and 20. If the final result for the current machine is 5, you would then store the trial expiration details in the Registry key designated by ArrayTrialEntries[5]. In this scenario, although a cracker might discern that a Registry entry is being used to store trial information, releasing a crack to delete that specific Registry key won't be universally effective, only impacting computers where the computed machine number equaled 5.

Utilize the SecureEngine protection macros to secure all code that chooses and accesses the Registry Keys, and maintain the encryption of all strings. This approach will make it substantially harder for a cracker to decipher the process you're employing to select one Registry key over another.

3.9 Implement Your Unique Ideas

Employing a software protection (like Themida, WinLicense or Code Virtualizer) can be an effective strategy to guard against cracking attempts, especially when used judiciously. Enhancing this with your original protection strategies will further bolster the defense of your software.

You don't need to be a security expert or have knowledge about cracking techniques to fortify your application. Understanding your application intimately and incorporating additional safeguards to shield sensitive code sections is key. For instance, a cracker's primary aim would be to activate the "registered mode" of your application without a valid license. They generally aren't interested in other facets, only in enabling the registered mode. Given this, you could

intensify security by implementing supplementary verification steps within different routines, such as:

- Confirming the presence of a legitimate license file.
- Utilizing a secondary function that also scrutinizes the license.
- Ensuring that data extracted from the license file conforms to the expected format.
- Applying an internal CRC to verify the accuracy of license data, among others.

By integrating numerous small additional verification steps into your source code, you elevate the level of security within your application. Moreover, safeguarding these checks with the VIRTUALIZER macros will significantly complicate the task for crackers.

Part

4 Contact us

If you want to share with us some of your protection ideas to be included in this guide or have any questions, we are happy to hear from you. You can contact us at support@oreans.com