FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Protecting Sensitive Data in Memory in .NET
Student:	Bc. Viktor Dohnal
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

1) Study security issues inherent in memory protection, e.g. leaking of sensitive information into untrusted environments.
2) Research memory protection techniques available in .NET framework, e.g. SecureString.
3) Analyze how these techniques are implemented in various environments (Win32, Win64, Mono...).
4) Prepare a proof-of-concept of an application that contrasts the confidentiality of sensitive data (e.g. a password or an encryption key) when implemented using traditional .NET functionality and when using a more secure version using the more advanced techniques.
5) Analyze the real-world security of these protections: Are they being used in real applications? Can an attacker overcome them?
6) Discuss your results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 31 January 2023 in Prague.

Master's thesis

PROTECTING SENSITIVE DATA IN MEMORY IN .NET

Bc. Viktor Dohnal

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Josef Kokeš, Ph.D.
May 4, 2023

Contents

List of Figures

List of code listings

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 4, 2023 .....

Abstract

Applications frequently handle sensitive information, such as passwords and encryption keys, which are typically stored in volatile memory alongside other data. This thesis investigates the efficacy and implementation of memory protection techniques within the .NET ecosystem. The findings have been applied to the analysis of the KeePass password manager, which led to a vulnerability discovery. The vulnerability allows an attacker to recover the master password from memory, even when a workspace is locked or KeePass is no longer running.

Keywords Memory protection, sensitive data, .NET, SecureString, KeePass

Abstrakt

Aplikace často pracují s citlivými informacemi, jako jsou hesla a šifrovací klíče, které se obvykle ukládají do operační paměti spolu s dalšími daty. Tato práce zkoumá účinnost a implementaci technik ochrany paměti v ekosystému .NET. Získané poznatky byly aplikovány na analýzu správce hesel KeePass, která vedla k odhalení zranitelnosti. Tato zranitelnost umožňuje útočníkovi obnovit hlavní heslo z paměti, i když je aplikace uzamčena nebo KeePass již není spuštěn.

Klíčová slova Ochrana paměti, citlivá data, .NET, SecureString, KeePass

Acronyms

BSTR	Basic string or binary string (Data type used by Windows)
CLR	Common Language Runtime
GC	Garbage Collector
TPM	Trusted Platform Module
UWP	Universal Windows Platform
WinAPI	Windows API
WinForms	Windows Forms
WPF	Windows Presentation Foundation (GUI framework)

Introduction

The internet, and by extension, a significant portion of the global economy, relies on certain values, such as passwords, remaining a secret. Without this fundamental capability, no social network, bank, or e-commerce site would be able to function. And it's not just passwords: API tokens, credit card numbers, or encryption keys are all expected to be kept secure and private in order to maintain the trust and integrity of online transactions and communications. Making sure that such secrets don't fall into the wrong hands has become a constant struggle for security professionals across the industry.

Whether or not storing these values unprotected in the program's memory is a security issue has long been debated among engineers. One perspective is that, in the event an attacker gains access to the application's memory, further resistance would be futile, and the attacker would have already achieved their goal. On the other side, there are defense-in-depth principles and the idea that relying on one layer of security is never enough. In the real world, the attacker is unlikely to have full access to everything at any given moment, but instead, access to memory may exist only under specific conditions and affect only specific regions.

Historically, **the importance of security was not fully recognized** in the design of computers, protocols, or entire IT ecosystems. This was partially due to the belief that people who owned computers, which were expensive pieces of equipment at the time, were professionals who were unlikely to cause harm. This was true even for national security-related matters, such as the military. When in 1984 [1], Bill Landreth, a young hacker, was found to gain access to United States department of defense files illegally, the FBI had trouble identifying a law that was broken. The reason was that, at the time, there were no laws covering "computer trespassing" - unlawful access to computer systems. Landreth got charged with wire fraud, the closest fitting offense at the time.

While 1984 might sound way back in the past, personal computing was already beginning to pick up steam, with Macintosh being introduced that year. And enterprises have been using computers for many years by then. For example, Microsoft DOS, on which today's Windows OS is based, was introduced in 1981. As it was focused on simple personal use, it didn't support any form of isolation, user accounts, or access permissions. Combined with a strong focus on the **backward compatibility** of market leaders such as Microsoft, many of the hardware and software design decisions from that time still influence how the technology we use today works. And if systems at the time were insecure, we can expect our systems to be built on insecure fundamentals as well.

While with each generation, security generally improves, it is a complicated process since backward compatibility requirements constrain engineers. Instead of redesigning a whole part of the system, more often, an **additional layer** is introduced on top of it. A good example would be data and code being stored in the same memory space, unseparated (von Neumann architecture). Apart from a few microcontrollers, this is the way computers have been designed for decades

now. While it undeniably has many benefits, it is also a cause of numerous security issues. A common example would be buffer overflow attacks that result in code execution. Attempts at fixing the issue can never tackle the fundamental problem and instead have to pivot towards adding new features on top of existing solutions. Indeed, measures such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), or stack canaries only make it harder for an attacker to exploit a vulnerability but cannot prevent it entirely.

This phenomenon is closely related to the topic of this thesis. Major computer architectures used today weren't designed to support a use case where sensitive and non-sensitive values are completely separated in memory in a way that is guaranteed to prevent everyone but the authorized application from accessing that information. This is also true for applications that need the capability to conduct cryptographic operations but don't need access to the keys themselves. As with the previous example, additional capabilities were added over time that attempted to help mitigate the issue. The goal of this thesis is to explore these capabilities broadly and then in more detail for the .NET programming language.

Structure and Goals of This Thesis

The goals of this thesis, together with chapters focused on the topic, follow.

- **Protection of sensitive data in memory in general**
  Describe the scope of sensitive data that needs protecting, how it can leak, and what the protections are on different levels, like the operating system or a programming language. This is covered in Chapter 1, Sensitive Data in Memory.

- **Memory protection in .NET specifically**
  Investigate the different memory protection techniques available in the .NET framework, such as `SecureString`, across different operating systems and runtimes. Create proof-of-concept applications that demonstrate the efficacy of these protections. This is covered in Chapter 2, Sensitive Data in .NET.

- **Analyze the real-world usage**
  Reverse-engineer memory protection in a password manager written in .NET and describe how it works. This is covered in Chapter 3, Real-World Usage.

Chapter 1

Sensitive Data in Memory

1.1 Sensitive Data - Scope

The subsequent analysis will delve into data categories that are particularly susceptible to memory leaks. Leading among these categories are login credentials, comprising a username and password, which are typically the most conspicuous type of data at risk of compromise. If a desktop application needs credentials to communicate with a server, it likely needs to store them in memory, at least for a short period of time. It might also support some form of auto-save feature, where the credentials are pre-filled for a user, so they only need to be entered once. This means the password also has to be stored in persistent storage (out of the scope of this thesis) and then loaded into memory. **Password managers** offer credential management as their main feature, so their creators need to be especially aware of potential security implications. This includes password managers embedded in other applications, such as web browsers. Credentials aren't just used in client-side applications. On the server, the application is likely to require login credentials to access other services, such as a database server.

Login credentials might also come in the form of **API access tokens**, random strings generated by the service provider. They are used to authenticate an application before accessing a service, such as a social network API. Also, services related to CI/CD, such as source code management systems (SCMs), build servers, and code scanners, are usually accessed by using these tokens. They are less common for the client-side software but can be found there as well. An example would be trading software that needs access to stock exchange data. Stealing this value means an attacker has the same access to the API, just like the original process.

For client-side software, it is more common to see JSON web tokens (JWTs) or session IDs. They are generated on the server, typically during authentication, and sent to the client. The client then sends this data to the server with every request. On the client, these are usually stored as **cookies**, either in web browsers or desktop applications based on them (e.g., Electron). If this value leaks, an attacker can impersonate the user and generally do whatever the user is capable of doing with the service.

A category closely related to passwords and API access tokens is **cryptographic secrets**. An obvious use is case encryption and decryption, which require keys. Disk encryption software, such as Bitlocker or Veracrypt, must keep encryption keys in memory to decrypt and encrypt data on the fly. A leaked encryption key results in an attacker being able to decrypt the contents of the disk or craft malicious content without the user being able to detect it. The situation is similar to any other software dealing with cryptography, such as an email client that uses OpenPGP to decrypt messages or sign them. On the server, probably the most widespread use case is TLS certificates. They are used to encrypt HTTP traffic with the client (HTTPS). In order for this to work, the server needs to hold the corresponding private key in memory. This

value can be precious to attackers because, with it, they can read this traffic or even change it with man-in-middle attacks. Another example is keys that are used to encrypt or decrypt data in use, transit, or at rest. Data that require encryption are often encrypted because of regulatory requirements (such as patient information), so the keys might be interesting for bad actors. A special sub-category that also needs to protect cryptographic secrets is DRM solutions. If secrets leak from memory, the whole system can be compromised and bypassed.

Other sensitive data, such as **personally identifiable information** or **financial data**, are also potential targets. For example, if a server processes credit card numbers, an attacker with access to memory might be able to obtain this data.

To summarise, any data an attacker could misuse should be treated with caution when working with it in memory. The following subchapter looks at how this information could leak.

1.2 Exposure Possibilities

In the preceding subchapter, an analysis was conducted to identify potential data leaks and their associated consequences. In this section, the focus shifts towards investigating the actual occurrence of such leaks.

One possible source of leakage, which may not immediately come to mind, is **accidental** disclosure, popularized by OWASP's TOP 10 "sensitive data exposure" [2]. Sensitive data might simply be outputted to a **log file** or standard output/error. It can be easier for an attacker to gain access to such output because it is hard to keep track of logs in a large environment, and thus, they might be less protected. Furthermore, if an application is built in a debug mode, it is more likely to print or log detailed debugging messages. The developer might forget to turn off debugging mode before deployment. Sensitive data can also be a part of an error message (especially web servers are guilty of this). Programming languages aim at making engineer's life easier by making the default behavior desirable in most situations. Unfortunately, the unintended consequence is that it is too easy to print a password or an encryption key to a log file by mistake. Furthermore, the default serialization of objects in many high-level languages means that if a class or any of its parents contain sensitive data, the serialized object will also contain it. Ideally, a software engineer should be prevented by the programming language from unintentionally exposing sensitive data or at least receive a warning.

Another way a leak can occur is related to the first category. Applications and operating systems can generate files that contain the whole application's memory or its part. A **crash dump** is a file that contains information about a program's state at the time it crashed. Depending on the programming language and operating system, it can, for example, include CPU registers, stack, heap, and the particular instruction being executed at the time of the crash. Other parameters, such as the level of protection, location of the dump on the filesystem, or who is responsible for its creation (OS or the application), also vary. Many Linux distributions create crash dumps (there called core dumps) by default in the same directory as where the executable is. This can be problematic because the user (administrator) now has to remember to either disable core dumps altogether, change their default location to somewhere more secure, or tighten the permissions of these files so that access to resulting core dumps is restricted (by default, the owner of the file is the same user that ran the crashed process). And even then, these crash dumps could still leak.

The code in the code listing (1.1) serves as a concrete example. It prompts the user for a secret string and then crashes, which results in a core dump being created by the OS.

Code listing 1.1 A simple C++ program that accepts a secret value (password) as standard input and then crashes

```
#include <iostream>

int main() {
    // Simulate a secret value in memory
    char password[20];
    std::cout << "Enter␣password:␣";
    std::cin >> password;

    // Crash the program - segfault
    // Coredump generated
    int* null_pointer = nullptr;
    *null_pointer = 0;

    return 0;
}
```

The code was then compiled with `g++ -O0 main.cpp -o main`. Note that the `-g` flag wasn't used. This behavior doesn't depend on a program being compiled in debug mode. After running the program on GNU/Linux Debian 11 (Bullseye), it is easy to verify that the crash dump contains the secret string. The attacker doesn't even have to be familiar with the crash dump's structure or the program's memory layout. Running `strings` on the file and then looking for values that resemble credentials might suffice, as demonstrated in the listing 1.2. For a more detailed analysis, the attacker could load the core dump with a tool like `gdb`.

Code listing 1.2 Demonstration of the potential simplicity of extracting sensitive data from core dumps

```
$ ./main
Enter password: secret-string-123
Segmentation fault (core dumped)
$ strings core | grep secret
secret-string-123
secret-string-123
```

Therefore, when assessing crash dumps of a given programming language and target operating system, it is essential to understand what exactly is included in the crash dumps and how hard they are to access.

While crash dumps only contain a memory of a particular application, there are other file types that contain much more, sometimes the whole memory. **Hibernation**, or "safe sleep" on macOS, is a power-saving state that saves the contents of a computer's memory to disk, allowing it to be powered off entirely. It is a common feature of desktop and laptop computers. On macOS, the hibernation file is called `sleepimage` and is stored in the `/private/var/vm/` directory. On Linux, the hibernation file is, by default, stored in a separate partition called a "swap partition." On Windows, the hibernation file is called `hiberfil.sys` and is located in the root directory of the system drive.

Swap allows the operating system to use hard disk space to store data that would otherwise be kept in RAM. When the system runs out of physical memory, it moves some of the least recently used data from RAM to the swap space, freeing up the RAM for other uses. On Linux, swap space is typically allocated as a separate partition on the hard drive; on Windows and

macOS, it is stored in a file. The file is `pagefile.sys` on Windows, stored in the root directory of the system drive, and a group of files in `/private/var/vm/` named `swapfile` with a number at the end on macOS. Inspecting these files is more challenging than inspecting hibernation files because the memory might not be continuous (pages are scattered). However, since especially credentials are usually tens of bytes long (AES keys take up 16 or 32 bytes) and on modern operating systems, page sizes are in kilobytes, the attacker isn't likely to be severely limited by this.

Both swap and hibernation files are potentially more valuable for an attacker than a crash dump since they contain data from the whole OS, not just one application. Compared to crash dumps, they are harder to access on a live system because of tighter access control (administrator/root privileges are required by default for read access), but the consequences can be severe. As shown in [3], it is relatively easy to dump Veracrypt (disk encryption software) key from a Windows hibernation file. Another strong point is that the data can remain available long after the application has been shut down, both in the swap and hibernation file. So even credentials or sensitive data that were never directly saved to a hard disk by the application directly can end up being stored there.

Vulnerabilities in the code can also lead to data leaks. An attacker might exploit vulnerabilities and trick the program into exposing memory that developers haven't intended to be exposed. A good example is a buffer over-read vulnerability named Heartbleed that was discovered in the OpenSSL library. The bug allowed attackers to exploit a vulnerability in the OpenSSL heartbeat extension and read sensitive information, such as cryptographic keys, from the memory of the affected systems. The vulnerability was introduced in OpenSSL 1.0.1, which was released in March 2012, and it went undetected for over two years until it was discovered and publicly disclosed in April 2014 [4]. The bug was present in the code because of a missing bounds check in the implementation of the heartbeat protocol, which allowed attackers to send specially crafted heartbeat requests to the server and receive data from the server's memory. An attacker can read 64 kilobytes of process memory for a single Heartbeat message. By repeating this enough times, attackers were able to extract the memory of the entire process.

Even when an application doesn't contain any vulnerabilities like those described, it might still get its memory exposed. The security of the application ultimately depends on how all layers below it, like the operating system or the underlying hardware, are secure. Permissions might be misconfigured, or there could be a vulnerability in the operating system itself, allowing the attacker to dump memory regions (or an entire process). If the attacker compromises a virtualization host, allowing them to dump the memory of an entire virtual machine, it's difficult for an application inside that VM to protect its memory. In addition to vulnerabilities in code, hardware vulnerabilities such as Spectre and Meltdown allow attackers to leak memory from an application. They work by exploiting the speculative execution feature of modern CPUs, which is used to improve the performance of the CPU. The vulnerabilities allow attackers to access privileged memory, such as kernel memory, from the memory of the affected systems. Such **side-channel** attacks are even more challenging to defend against since, at the application level, they can't even be detected.

If an attacker has physical access to the device, it might be possible to attack the Direct Memory Access (DMA) capability [5]. As the name suggests, it's a hardware capability to allow direct access to memory through interfaces such as FireWire, Thunderbolt, or ExpressCard. If a computer is running and vulnerable to the attack, it can be used to create a memory dump, even when the computer is password protected and uses full disk encryption. The encryption keys can then be found in the memory dump. This is often used by forensic tools.

Vulnerabilities don't have to be at a lower level to cause sensitive data exposure. **Cookies** often contain session tokens, which are essentially credentials used to authenticate users against a server. If a website is vulnerable to Cross-Site Scripting (XSS), an attacker has the ability to execute malicious Javascript on it, for example, by sending a crafted link to the user. When the user clicks the link and visits the website, the attacker's Javascript is executed. Since the

user is already authenticated and the cookies are present, the code steals these cookies and sends them over to the attacker's server. Since this is a well-known attack, there are measures developers can take to make exploitation harder. A relevant example for this thesis would be setting the cookies as HTTP-only. When a cookie is flagged as HTTP-only, it means that it can only be accessed and modified by the server and not by client-side scripts running in the user's browser. This is enforced by the browser. Unfortunately, this protection doesn't work when an attacker has access to the browser's memory or its files on the disk. Since the data is unencrypted and without any extra protection, malware authors took notice. The RedLine Stealer [6] focuses on stealing cookies from web browsers so that the attacker can then use them to act as the compromised user. Stealing session cookies is a devastating attack because it also bypasses 2nd-factor authentication.

In summary, there's a wide range of memory exposure possibilities an application developer has to be aware of. The following subchapter explores the possible protections on different levels of computer architecture.

1.3 Protection Measures

As explored previously, the security of an application depends not only on the application itself but also on the security of all the levels (architecturally) below it, such as the OS or the hardware. This subchapter explores available memory protection mechanisms on all these levels, with a summary of tools an application developer can use to detect issues.

1.3.1 Operating Systems and Conventional Hardware

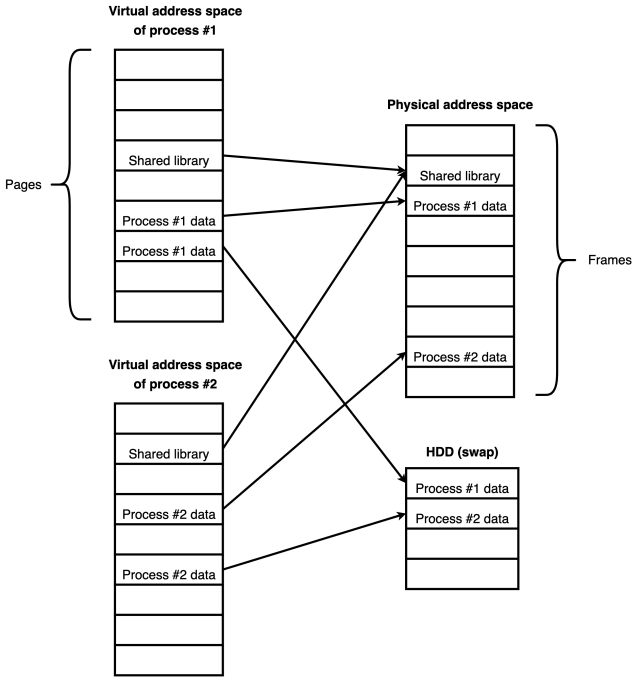
Modern operating systems do not allow regular processes to allocate physical RAM. Instead, they utilize **virtual memory**. Virtual memory is a technique used to create an illusion of a continuous memory block for every process. The block can also be larger than the amount of physically available memory. It also allows the operating system to allocate virtual memory to different processes without having to allocate physical memory, which can be allocated by the operating system only when the process actually needs it. The physical memory that's not being actively used can be stored on the hard drive through a process called swapping. Swapping can also be used if a process needs more physical memory than is currently available, where the additional requested memory is stored on the hard drive as well.

Virtual memory is managed through a process called **paging**. Process's virtual memory space is divided into small parts called pages, and these pages are then mapped by the OS to either physical memory frames (page size equals frame size) or pages on the hard drive. The information about the mapping is stored in the **page table**, and each process has its own. In order to make the translation process more efficient, support for virtual memory is built right into the hardware. The CPU works with virtual addresses natively and utilizes the memory management unit (MMU) component for virtual-to-physical address translation. The operating system is then responsible for handling page faults, errors that arise when the MMU cannot translate the virtual address to a physical address, and for mapping pages to their hard drive storage.

Pages can be mapped to the same frames between processes to save memory. For example, a shared library can be used by multiple processes at once. To prevent a malicious process from changing this shared memory for all other processes, copy-on-write protection was introduced. This mechanism allows multiple processes to share the same memory pages until one of the processes modifies the page. When a process modifies a shared page, the page is copied, and the process gets its own private copy of the page.

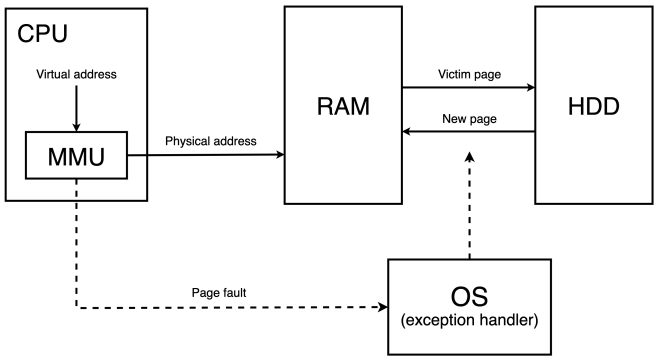
Virtual memory is further divided into **kernel space** and **user space**. They are commonly implemented using protection rings, which is a concept describing how code can run with different

Figure 1.1 Memory virtualization in modern operating systems



privilege levels. Protection rings are natively supported by CPUs, and only the operating system can switch between them. For example, in x86 architecture, the kernel or operating system runs in ring 0, with full access to all system resources. This includes access to system memory, I/O ports, and hardware devices. There are also specialized instructions and registers that can only be used or accessed in ring 0. Ring 1 and ring 2 are not typically used in modern operating systems, but they were originally designed for use by device drivers and other low-level system software. User-level applications and programs run in ring 3 with limited access to system resources. In a modern operating system, if a user-mode application wants to access privileged resources, it needs to ask the operating system through a **system call**. The operating system then executes the operation and provides the result back to the application, if there is any.

Figure 1.2 The process of translating a virtual address to a physical one



Memory virtualization is a fundamental capability behind **process isolation**. However, it is important to state that even with its existence, **processes running under the same user usually have read/write access to each other's memory spaces** by default. This is because operating systems provide mechanisms for doing so. For example, in Windows, a process can utilize WinAPI function `ReadProcessMemory`, as shown in the example 1.3. However, if a process is running under a different user, this will not work (by default). Of course, as one would expect, it is possible to create a memory dump of the entire system with administrator privileges.

Code listing 1.3 A simple C# program that obtains necessary permissions and reads memory from a different process

```
public class ReadProcessMemory
{
    [DllImport("kernel32.dll")]
    public static extern IntPtr OpenProcess(int dwDesiredAccess,
                                bool bInheritHandle, int dwProcessId);

    [DllImport("kernel32.dll")]
    public static extern bool ReadProcessMemory(int hProcess,
                      Int64 lpBaseAddress, byte[] lpBuffer, int dwSize,
                      ref int lpNumberOfBytesRead);

    public static void Main()
    {
        int PID = 1234 // Target process ID
        Int64 address = 0x12345 // Target address in the process
        int bufferLen = 123

        int bytesRead = 0;
        byte[] buffer = new byte[bufferLen];

        IntPtr processHandle = OpenProcess(0x0010 /*PROCESS_WM_READ*/,
                                           false, PID);
        ReadProcessMemory((int)processHandle, address, buffer,
                                         buffer.Length, ref bytesRead);

        // Process memory is now in the buffer variable
    }
}
```

When a block of memory containing sensitive data isn't used anymore, it should be deleted immediately. That's why most languages and operating systems offer functions dedicated to this purpose. It is not a good idea to use general-purpose functions because those could be optimized away by the language runtime. For example, on Windows, one can use `SecureZeroMemory`, to always overwrite a passed memory block with zeros. On macOS or Linux, `memset_s` can be used instead.

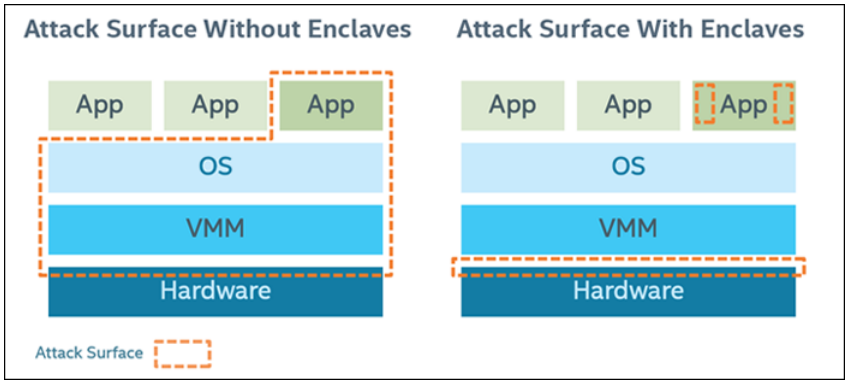
Some of the aspects discussed so far can be changed with permissions and privileges settings. In modern operating systems, this is achieved with **Access Control Lists (ACLs)**. They are used to regulate access to almost any system resources, from files and directories to devices and processes. These access rights can be granted or denied to individual users or groups or to a combination of both. They can also be inherited. While they aren't directly related to the protection of sensitive data in RAM, they are an important aspect of general application security. A process can also give up privileges it doesn't need, which is an important capability

when following the least privilege principle. For example, instead of always having access to LSA secrets (more below) from the main process directly, a developer can choose to delegate this privileged operation to a separate application while limiting the privileges of both applications to resources they truly need. This way, even if the main application is compromised or its memory leaks, the impact is smaller. ACLs are also useful for adequately protecting crash dumps, hibernation files, swap files, and other instances of RAM being saved to the filesystem.

1.3.1.1 Intel SGX

There have been attempts at adding hardware support for memory isolation into x86 CPUs, out of which the most notable is **Intel Software Guard Extensions (SGX)** [7]. It was first introduced in 2015 with the launch of Intel Skylake processors. Intel SGX is a hardware-based security technology that provides a secure enclave within a processor, allowing user-space applications to execute code and store data in a trusted environment that is protected from other software on the system, including the operating system itself. In theory, even if the operating system or the hypervisor is compromised, data in the secure enclave should remain safe.

Figure 1.3 Diagram showing the difference between potential attack surface of an application storing sensitive data in RAM and an application using a secure enclave (Intel SGX) [8]



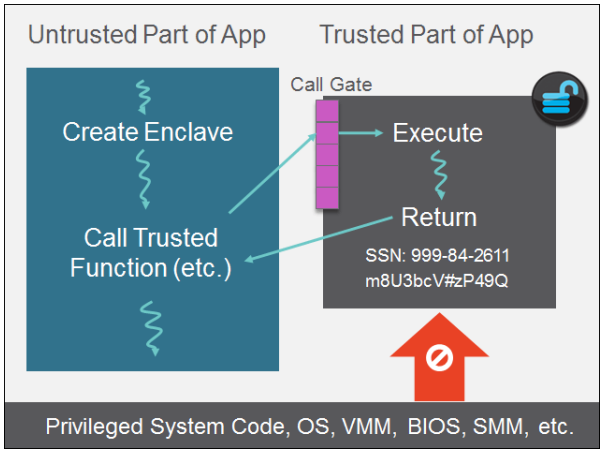
First, the application (insecure part) creates an enclave (secure part) by providing data and code to be executed. From that point on, only the code running in the enclave will have access to that data, so not even the application itself can access it. The application can then communicate with the enclave by calling functions it had previously defined (functions are called using a special instruction). For example, it could call an enclave function to sign some data and retrieve the signature, while the private key would never leave the enclave.

SGX utilizes the Enclave Page Cache (EPC), a protected memory region used by the CPU during the enclave's code execution. This region is encrypted and decrypted with Memory Encryption Engine (MEE), a dedicated chip that also holds the symmetric encryption key. This key is generated and placed into MME registers during system boot and erased during the shutdown, which effectively destroys the enclave too. However, it is possible to keep the same enclave across boots by sealing it - securely storing it on the HDD encrypted and loading it back the next time the application needs it. Encryption is performed using a dedicated key (Seal Key) that's only available internally to the CPU.

Using SGX can undeniably benefit security. If an attacker has only access to a memory dump (e.g., through a vulnerability) but isn't able to execute any code on the machine, secrets in the enclave's memory should remain protected. Even DMA vulnerabilities that can allow access to all of the system's RAM are unlikely to work because the enclave's memory is encrypted, and the key for its decryption is stored in the CPU and never in RAM. Since its inception, SGX has been integrated into many different applications. Digital Rights Management (DRM) technologies,

such as the one used to play Blu-ray disks, rely on SGX to keep keys used to decrypt the content from leaking. Some password managers, such as 1Password, use SGX to store master keys in an enclave [9].

Figure 1.4 Diagram from Intel summarizing the separation between application code and SGX enclave code [10]



However, various drawbacks prevented the wider adoption of the technology. While it addresses some security issues, others remain unresolved. The first point has to do with its architecture in general. Suppose that the enclave decrypts and returns the requested password. Then attacker's code can request that password as well. Furthermore, there are numerous vulnerabilities that allow an attacker to extract secrets from the enclave, such as susceptibility to side-channel timing attacks [11]. It was also demonstrated that malware running in an enclave in one virtual machine could steal data from an enclave running in a different virtual machine on the same system [12]. This is especially concerning for users of cloud services. Some of these vulnerabilities proved rather difficult to patch.

Another of SGX's drawbacks lies in its licensing and development. Before developing an application utilizing SGX, a developer first needs to agree to Intel's license terms and then obtain a certificate for signing the enclaves [13]. The CPU will refuse to load the enclave unless its signature is verified and the signer is found on the Intel SGX whitelist. This can be an issue for open-source developers. Furthermore, using SGX correctly requires knowledge of low-level languages and related security principles, increasing the cost of development.

Possibly because of the discussed drawbacks, in 2021, **Intel deprecated SGX** and removed it from the 11th and 12th generation Intel Core Processors. This means that users of those or newer Intel CPUs can no longer play Blu-ray disks since the DRM technology depended on it. Development continues on the Intel Xeon series focused on data center use.

1.3.1.2 Intel SGX Alternatives

While technology from AMD, **Secure Encrypted Virtualization (SEV)** [14], also focuses on the problem of memory protection, it has different goals than SGX. As the name suggests, it allows the memory of a virtual machine to be encrypted with a unique key that's not available to the host OS, which effectively protects the VM guest from a compromised host. This, together with the ability of the guest OS to verify that its RAM hasn't been tampered with, makes it an interesting proposition for cloud providers and their customers. SEV is currently supported only by the AMD EPYC series of CPUs, which is targeted on server deployment. It is not available in any consumer series CPU, such as RYZEN, so it is unlikely a consumer application would utilize this feature.

What if there's no hardware support for secrets outside RAM? The paper "**TRESOR** Runs Encryption Securely Outside RAM" [15] proposes to use CPU registers as key storage instead. It was originally aimed at preventing the cold-boot attack, which consists of capturing a powered-on computer, cooling its RAM sticks, and then connecting them to the attacker's computer. Since RAM decays slower when cooled down, an attacker might be able to recover the original content. Even when full disk encryption is used, if the encryption key is stored in RAM, it could be broken this way. TRESOR prevents this by encrypting the RAM with a key that's stored outside of it. It uses x86 debug registers as key storage and AES-NI or SSE2 for RAM encryption and decryption.

While TRESOR can protect against basic memory dumps, it suffers from vulnerabilities, such as DMA attacks (injecting code that dumps the registers), and cannot be used as trusted key storage. Another drawback of this approach is that debug registers can't be used anymore. It also shouldn't be considered as a replacement for AMD SEV for protecting VMs memory since the CPU registers can easily be accessed by the host OS. It then comes as no surprise that TRESOR hasn't seen mass adoption and wasn't implemented into the Linux kernel.

1.3.1.3 TPM

Trusted Platform Module (TPM) [16] is a hardware security component for storing secrets and providing cryptographic functions based on the Trusted Computing Group open standard. It can be a standalone chip, or it can be integrated into a CPU package with its own dedicated memory. The main purpose is to provide an isolated environment that protects the stored secrets even if an attacker has access to the system's RAM. It has a cryptographic root of trust that is designed to never leave the TPM, which is used to protect secrets.

TPM is used in a variety of ways, but unlike Intel SGX, it **cannot be used to run custom code**. Instead, its main focus is to provide a small and simple group of functionalities that reduce the attack surface and are easier to audit. The first and most obvious use case is the storage and management of cryptographic keys. The TPM can securely generate and store cryptographic keys and provide cryptographic operations such as encryption and decryption. Apart from that, its storage can also hold other secrets like passwords and certificates. It can also be used to ensure the integrity of the system boot process by verifying that the system hasn't been tampered with. This feature is called Secure Boot, and it works by verifying the signature of the system that's about to be booted.

Software communicating with the TPM, be it the operating system or an application, must authenticate itself. This can be done in multiple ways, including using an authentication token, password, or **Platform Configuration Registers (PCRs)**. When the computer boots up, the TPM performs a series of measurements to establish a baseline for the system's configuration. These measurements are stored in PCRs. The TPM also contains a Root of Trust for Measurement (RTM) which is a cryptographically secure way to establish the authenticity of the boot loader, operating system, and other system components. By using RTM and verifying the state of PCRs, it is possible to release the encryption key and boot from an encrypted HDD up to the Windows login screen without the user providing any passphrase or encryption keys. If an attacker tried to boot a different operating system on the same hardware, these checks would fail, the key wouldn't be released from the TPM, and the HDD would stay encrypted.

When a key is generated by the TPM without the intention of ever leaving it, it can be encrypted by the TPM and saved to permanent storage like an HDD. This process is called sealing or wrapping, and it guarantees that the key can only be decrypted and used by that same TPM. When an application needs to conduct operations with that key again, it can simply let the TPM load it. In some instances, this can be more secure because the key cannot be extracted by an attacker during the time an application isn't using it.

TPM can also be used to **prevent brute-force attacks**. Without a TPM, if an attacker is able to get access to encrypted data or keys, they can try to guess the passphrase that was

used for the encryption. A TPM, however, might stop responding after a certain number of tries or even delete the secrets entirely. For example, if an attacker obtains an image of an encrypted HDD, it is possible to try passwords one by one until success. If the encryption key is stored in the TPM, the disk image alone won't help the attacker much because now they need to brute-force the cryptographic key (which is usually random) instead of a passphrase.

While using a TPM has undeniably many benefits, there are also drawbacks. First, the TPM can contain vulnerabilities, which for TPMs certainly aren't rare [17], [18]. And even if the TPM itself isn't vulnerable, it might be possible to monitor the traffic to and from the chip. Researchers were able to demonstrate capturing HDD encryption keys during boot and bypassing Bitlocker encryption entirely [19]. Of course, this is only an issue if the TPM is a standalone chip.

In most implementations, the TPM might not be powerful enough to handle large volumes of data with low latency. This comes as no surprise since it's often a simple microcontroller instead of a fully-fledged processor. Full disk encryption software might use the TPM to store the keys, but the on-the-fly encryption operations are still conducted by the main CPU. This means that not only is the HDD **encryption key stored in RAM**, but it can also end up in hibernation files and page files. For example, researchers demonstrated that access to memory dump in any of the forms mentioned is enough to obtain the volume key and **break Bitlocker** full disk encryption [20]. The same is true for other encryption software, such as Veracrypt [21].

1.3.1.4 Windows and DPAPI

Apart from Bitlocker and Secure Boot, **Windows** also utilizes the TPM in other system services and applications [22]. For biometric authentication, Windows Hello, and Windows Hello for Business, TPM is used to protect the authentication key. This allows the user to log in to Windows and authenticate against Active Directory services without using a password. Credential Guard is a new TPM-based feature targeted at businesses relying on Active Directory Domain Services. It is designed to protect user credentials by using virtualization to isolate the process that hashes them. It creates an isolated memory area that can only be accessed during the boot process and uses the TPM to protect its keys with TPM measurements (previously explained state of Platform Configuration Registers). This prevents malware from harvesting access tokens and using them to access other machines, a technique known as the "pass the hash" attack.

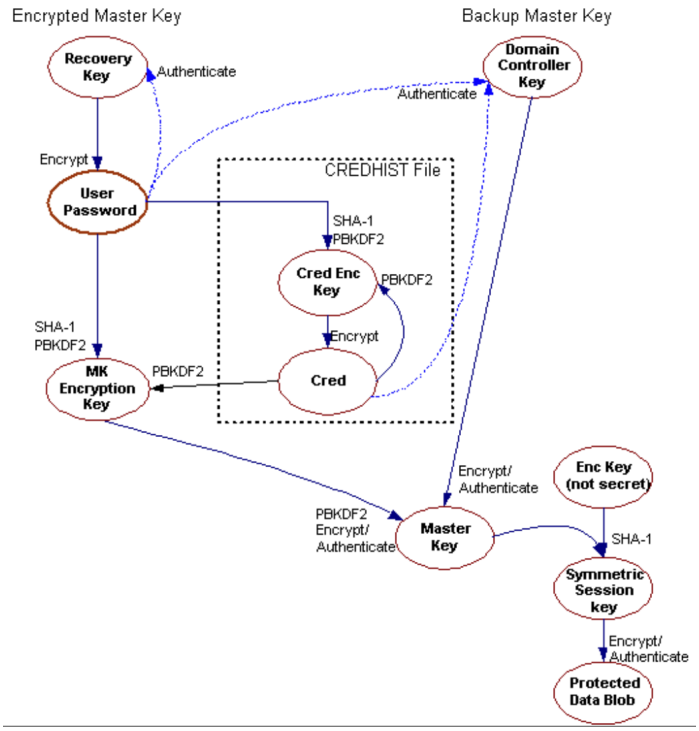
For low-level interaction with the TPM, developers can use TPM Base Services (TBS) API. For higher-level access, there's "Cryptography API: Next Generation" (**CNG**), available since Windows Vista. Microsoft also provides C# bindings for CNG [23] and a TPM toolkit [24] with simulator and abstraction interfaces for other languages, such as Java or Python. Despite this rich development ecosystem, it is rather difficult to find software, such as password managers, that claims to use these features to protect secrets. And if they do, it's usually related to announcing support for Windows Hello authentication, which internally uses a TPM [25].

What seems to be used on a much broader scale is **Data Protection Application Programming Interface (DPAPI)** [26], [27]. DPAPI is a cryptography-based API aimed at protecting sensitive data available in Windows, originally introduced in Windows 2000. It provides a simple interface that can be used to encrypt and decrypt data without explicitly providing an encryption key. Developers don't have to have to care about properly storing encryption keys, choosing a suitable algorithm with appropriate parameters, or ensuring ciphertext integrity. All of that is handled by the operating system. The main functions provided by the API are listed in table 1.5.

Figure 1.5 Windows DPAPI functions used for arbitrary data encryption and decryption. (`Dpapi.h`, `crypt32.dll`) [28]

DPAPI Function	Description
`CryptProtectData`	Performs encryption on the data in a `DATA_BLOB` structure.
`CryptProtectMemory`	Encrypts memory to prevent others from viewing sensitive information in your process.
`CryptUnprotectData`	Decrypts and does an integrity check of the data in a `DATA_BLOB` structure.
`CryptUnprotectMemory`	Decrypts memory that was encrypted using the `CryptProtectMemory` function.

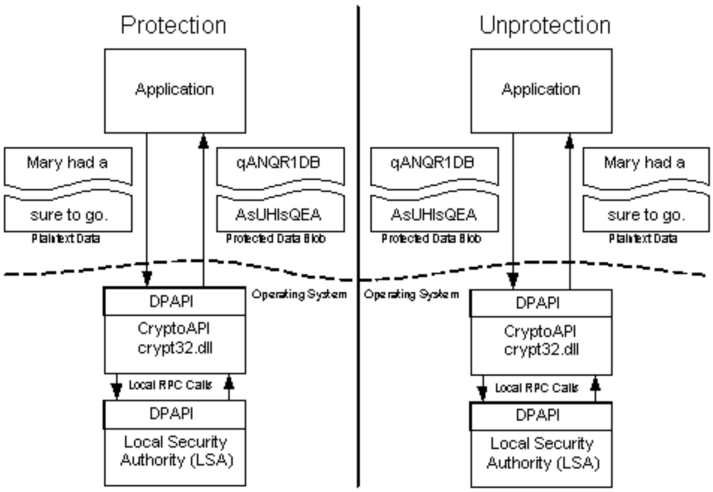
Figure 1.6 Diagram from Microsoft describing cryptography of the original DPAPI [26]



The original two functions available were `CryptProtectData` and `CryptUnprotectData`, so most of the available research focuses on them, but they are not as relevant to this thesis. They are used to encrypt data permanently so that they can be stored on the filesystem. While the particular cryptographic algorithms and their parameters have changed over the years, the architecture has remained nearly identical. The key used for the actual encryption operations is derived from the user's login credentials and other sources of entropy. It could be said that the security of these functions mainly relies on the security of the user's password, with other aspects being access control and isolation provided by the OS. Access control and isolation are provided by the Local Security Authority (LSA). It is running as a daemon (`lsass.exe`), and applications can use its RPC interface to gain access to the API 1.7.

CNG DPAPI (also called DPAPI-NG) [29] is a new version of DPAPI providing entirely new functions built on top of the CNG. Its main goal is to provide support for cloud and Active

Figure 1.7 Diagram from Microsoft describing high-level architecture of the original DPAPI [26]



Directory environments so that data encrypted on one device can be decrypted on another. However, it didn't replace the "old" DPAPI, which is still being used. Rather, it is meant to provide additional capabilities. This is unfortunate because, unlike the original DPAPI, in some cases, it uses a TPM to protect keys, making it more secure. On a system level, it is used in cooperation with the Windows Hello biometric authentication. Unlike the original DPAPI, DPAPI-NG doesn't even have dedicated C# objects to abstract away from calling WinAPI functions directly. Since it's not used in any of the .NET protections, it is out of the scope of this thesis.

DPAPI keys are stored outside swappable memory, so an attacker gaining access to the swap file (`pagefile.sys`) shouldn't be enough to break it. However, **the keys are still in RAM instead of a TPM, so if an attacker dumps all of it, DPAPI is compromised** [30]. This is true even for Windows 11. There are many offensive tools and attacks focused on both DPAPI and DPAPI-NG. On a running system with administrator access, tools like Mimikatz [31] can be used to dump DPAPI secrets and decrypt protected blobs; even dumping keys out of the TPM is supported for DPAPI-NG. DPAPI doesn't support encryption operations with custom or on-demand generated keys like Intel SGX, so it's impossible for developers to protect themselves against such attacks. If the operating system is compromised, so are the secrets. Also, it is **unclear what happens when the system is hibernated**. Are the keys cached in the hibernation file, on disk, or not cached at all?

Original DPAPI is used extensively both by Windows and also by 3rd party applications. Browsers like Chrome or Edge use it to protect password databases and cookies at rest. Windows relies on it to protect passwords, certificates, or EFS (Encrypting File System) keys.

`CryptProtectMemory` [32] and `CryptUnprotectMemory` [33] are much simpler and lighter functions that were introduced later. While they are technically part of DPAPI, most of the previously discussed characteristics don't apply to them. Their main purpose is to protect the memory of a running application. They support flags indicating whether the encrypted memory should be accessible only by the same process, different processes under the same user, or across the whole system. The encrypted data shouldn't be written to persistent storage, as the keys used to encrypt it will be lost once the user logs out. **There isn't much available research, so it is unclear where the keys are stored are how exactly the encryption works. The official documentation doesn't reveal any of this information.**

1.3.1.5 Reversing DPAPI

While the goal of this thesis isn't a thorough understanding of DPAPI, on Windows, .NET relies heavily on it. In some cases, official documentation is insufficient, and publicly available research doesn't go deep enough. As is later explored, some .NET classes call WinAPI functions `RtlEncryptMemory` (`SystemFunction040`) and `RtlDecryptMemory` (`SystemFunction041`). These semi-internal functions can be used to encrypt and decrypt memory without providing a key. Unfortunately, while some online articles assume these functions are related to DPAPI, none of them provide any evidence for this. Official documentation also doesn't specify how exactly the memory is protected or how these functions are related to official DPAPI interfaces. Without knowing what these functions do, the security of some .NET classes cannot be assessed.

Figure 1.8 Semi-internal WinAPI functions used by DPAPI to conduct crypto operations. Subject to change and not recommended for general use by Microsoft. (`ntsecapi.h`, `Advapi32.dll`) [34]

Function	Available as	Description
`RtlEncryptMemory`	`SystemFunction040`	Encrypts memory contents.
`RtlDecryptMemory`	`SystemFunction041`	Decrypts memory contents previously encrypted by the `RtlEncryptMemory` function.

Analysis of the `CryptProtectData` and `CryptUnprotectMemory` functions revealed that they are calling `SystemFunction040` and `SystemFunction041`. However, upon closer inspection, it was evident that they are not used for the actual encryption but rather as a security measure to protect data in transit during the RPC call to LSASS. An analysis of `CryptProtectMemory` and `CryptUnprotectMemory` showed that `CryptProtectMemory` and `CryptUnprotectMemory` are just wrappers around `SystemFunction040` and `SystemFunction041`. It is important to note that the behavior of the function might change without notice, as is clearly noted in the documentation. For verification of this discovery, a simple C++ program was written 1.5. It encrypts memory with `CryptProtectMemory` and then decrypts that same memory with `SystemFunction041`. Launching this program on Windows 7, 10, and 11 (both x86 and ARM) provided reasonable assurance that this is indeed the intended behavior.

Code listing 1.4 `CryptProtectMemory` code decompiled with IDA, after the `SystemFunction040` function pointer has been obtained by calling `ResolveDelayLoadedAPI`

```
__int64 dpapi_CryptProtectMemory()
{
  int v0; // eax
  unsigned int v2; // eax

  v0 = cryptbase_SystemFunction040();
  if ( v0 >= 0 )
    return 1i64;
  v2 = ((__int64 (__fastcall *)(_QWORD))ntdll_RtlNtStatusToDosError)
    ((unsigned int)v0);
  ((void (__fastcall *)(_QWORD))ntdll_RtlRestoreLastWin32Error)(v2);
  return 0i64;
}
```

Code listing 1.5 A simple Windows C++ program that verifies that memory encrypted with `CryptProtectMemory` can be decrypted with `RtlDecryptMemory`. Error checking ommited for brevity.

```
#include <windows.h>
#include <stdio.h>
#include <ntsecapi.h>
#include <ntstatus.h>

#pragma comment(lib, "crypt32.lib")

int main()
{
  // Prepare data for encryption
  const char* input_str = "Hello world of data protection.";
  const int input_len = 32;
  LPVOID protected_block = LocalAlloc(LPTR, input_len);
  memcpy(protected_block, input_str, input_len);

  // Tested for all 3 options
  int flag = CRYPTPROTECTMEMORY_SAME_PROCESS;

  // Encrypt memory
  if (CryptProtectMemory((LPVOID)protected_block, input_len, flag))
  {
    // Decrypt memory - different function!
    if (RtlDecryptMemory(protected_block, input_len, flag)
        != STATUS_SUCCESS);
    {
      if (memcmp((char*)protected_block, input_str, input_len) == 0)
        printf("SUCCESS: CryptUnprotectMemory == RtlDecryptMemory");
      else
        printf("ERROR: CryptUnprotectMemory != RtlDecryptMemory");
    }
  }
  LocalFree(protected_block);
}
```

The functionality of `SystemFunction040` warrants further investigation. It's sending an `IOCTL` to `\\Device\\KsecDD`. A further search of this topic yielded a code snippet on GitHub [35] that encrypts a block of memory with `CryptProtectMemory` and decrypts it by making the `IOCTL` directly. This means that the encryption keys are likely stored in kernel memory (it wouldn't make much sense to store them elsewhere). Also, due to historical reasons and other hints, it is quite unlikely a TPM is used to do the encryption operations or protect the keys. However, in order to prove this, kernel reverse engineering would likely be necessary, which is out of the scope of this thesis.

Just like `CryptProtectMemory`, `RtlEncryptMemory` supports three options on who should be able to access the encrypted data 1.8. While the official documentation discourages from using this function and recommends using `CryptProtectMemory` memory instead, as the next chapter shows, it is being used extensively. For further analysis, it can be assumed that the security level provided by `CryptProtectMemory` is identical to the one provided by `RtlEncryptMemory`.

Code listing 1.6 `SystemFunction040` decompiled by IDA and further simplified.

```
__int64 __fastcall cryptbase_SystemFunction040(__int64 a1,
    __int64 a2, int a3)
{
  __int64 v5; // rcx
  int v6; // edi
  int v7; // eax
  int v9; // ebx
  char v10[24]; // [rsp+50h] [rbp-18h] BYREF

  ... prepare parameters for IOCTL (omitted) ...

  // IOCTL
  return ((__int64 (__fastcall *)(__int64, _QWORD, _QWORD, _QWORD,
    char *, int, __int64, int, __int64, int))
           ntdll_NtDeviceIoControlFile)(v5, 0i64,
                                        0i64, 0i64,
                                        v10, v7,
                                        a1, v6,
                                        a1, v6);
}
```

Figure 1.9 `OptionFlags` parameter options for `RtlEncryptMemory` and `RtlDecryptMemory` functions. `CryptProtectData` has analogous options, named differently. [34]

Value	Meaning
`0`	Encrypt and decrypt memory in the same process. An application running in a different process will not be able to decrypt the data.
`RTL_ENCRYPT_OPTION_CROSS_PROCESS`	Encrypt and decrypt memory in different processes. An application running in a different process will be able to decrypt the data.
`RTL_ENCRYPT_OPTION_SAME_LOGON`	Use the same logon credentials to encrypt and decrypt memory in different processes. An application running in a different process will be able to decrypt the data. However, the process must run as the same user that encrypted the data and in the same logon session.

Finally, there's the issue of **hibernation**. In order to verify what happens to the keys, a simple test was done. A testing application periodically called the four DPAPI functions (`CryptProtectData`, `CryptUnprotectData`, `CryptProtectMemory`, `CryptUnprotectMemory`) in a loop. The program was started, and the computer hibernated. Once it was brought back from hibernation, it was left

on the lock screen for some time. After the user login, it was verified that the application was successfully able to use all four functions to encrypt and decrypt data during the time it ran behind the login screen. Since this required no user input, **it is clear that the DPAPI keys were cached on the disk during hibernation**. While more research would need to be done to know precisely, for simplicity, let's assume they were stored in the `hiberfil.sys` file like the rest of the system's memory.

1.3.1.6 Linux

There is no DPAPI alternative on Linux. Nor are there native system-wide interfaces that could be used to interact with a TPM easily. There are some 3rd party libraries, like the previously mentioned Microsoft TPM toolkit or tpm2-tss [36]. Intel SGX and AMD SEV are also supported on Linux.

Desktop environments Gnome, KDE, and their derivatives have their own userland key stores. Gnome has Gnome Keyring [37], and KDE has KWallet [38]. Neither of them uses any dedicated hardware (like a TPM), and they entirely rely on the security of the OS. They also unfortunately aren't making use of all available OS-level protections, such as SELinux, due to the lack of resources [39]. However, they attempt to mitigate some of the possible attacks, for example, by always running in a non-swappable memory [40].

Since there's nothing else, many popular applications use them, including commercial ones. For example, Microsoft Edge stores user passwords in either Gnome Keyring or KWallet [41].

1.3.1.7 macOS, Secure Enclave and Keychain

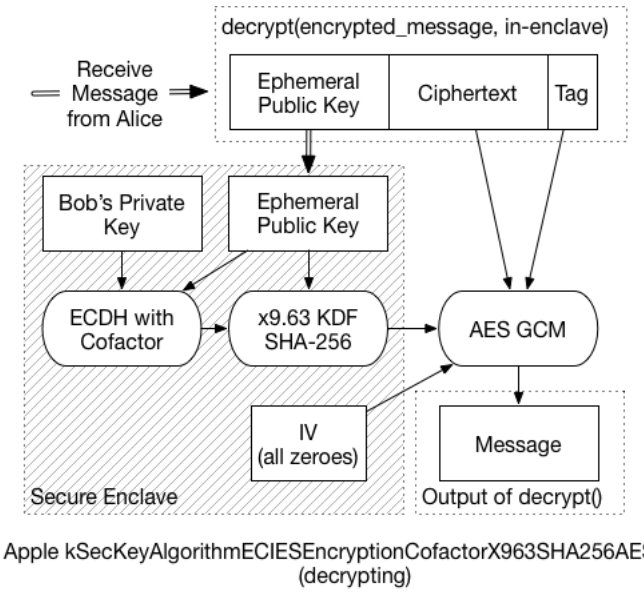
Neither Intel SGX, AMD SEV, nor a TPM is available on computers running **macOS**. Instead, Apple has developed a proprietary solution called **Secure Enclave** [42], which aims to provide similar functionality and is also available on other products from the same company. For desktop and laptop computers with Intel CPUs, it was implemented inside a separate SoC called T1 and later T2, which served multiple functions. These included a random number generator, storage controller for on-the-fly encryption, public-key accelerator, speech recognition, and also the previously mentioned Secure Enclave. The Secure Enclave itself contained multiple functions, out of which the main one was provided on an ARM-based processor called Secure Enclave Processor (SEP) running a dedicated operating system, sepOS. This processor uses regular RAM, but it has a dedicated memory region, and the communication happens only through another chip called Memory Protection Engine. Whenever the Secure Enclave requests a block of memory to be written, the Memory Protection Engine encrypts it and calculates an authentication tag, which is stored alongside it, together with a nonce to prevent replay attacks. Later, if either the tag or nonce doesn't match, the Memory Protection Engine signals an error to the Secure Enclave, which stops accepting requests until the system is rebooted. It also has dedicated nonvolatile storage for storing the generated keys and anti-replay tokens. The point of this hardware isolation is to protect the secrets, even in case the main OS kernel becomes compromised. Apple claims it is designed to be resistant to hardware side-channel attacks, such as differential power analysis. Since the introduction of the ARM-based M-series desktop and laptop CPUs, functions of the T2 chip have been incorporated into them with high-level architecture similar to T2 chips. Currently, the **Secure Enclave and application processor are integrated into the same SoC**.

The Secure Enclave is used by the operating system and pre-installed applications in various ways, many of which aren't accessible to 3rd party application developers. For example, it provides keys for the AES engine, which facilitates full disk encryption, and the memory protection engine, which encrypts RAM. Unlike with a TPM, the key used for full disk encryption doesn't have to be stored in RAM. It also processes biometric data (called Face ID and Touch ID) so that the operating system doesn't have access to them directly. The biometric data are provided directly by the biometric sensor, which is only connected to the enclave. The operating system

can only request biometric authentication and get a response on whether or not the data matches one of the previously saved representations or ask to store a new representation.

Application developers can use the enclave to generate keys and then ask the enclave to conduct cryptographic operations with them. The keys never leave the enclave, so not even the app developer has access to them. It also isn't possible to load pre-existing keys, as only new keys can be generated. This implies that if the device is lost, so are the keys, as they cannot be backed up. Currently, only NIST P-256 elliptic curve keys are supported. Apart from the expected signature, verification, and decryption operations, it is also possible to use the Elliptic Curve Integrated Encryption Scheme (ECIES). This is important because asymmetric cryptographic operations are much slower than symmetric, so for larger quantities of data, symmetric encryption is needed. It is based on the Elliptic-Curve Diffie-Hellman (ECDH) scheme for key exchange, with AES-GCM being the underlying symmetric cipher, which ensures confidentiality, authenticity, and integrity. Figure 1.10 demonstrates how decryption works. Note that the final decryption with the AES key happens outside the enclave. This is because the key is different for each message, so having access to it is equivalent to having access to the decrypted message itself, which by its nature, needs to be accessible by the application that requested the decryption in the first place. ECIES scheme can be used by developers for implementing secured communication with another service, as well as securely storing files on the local filesystem.

Figure 1.10 Message decryption with Apple Secure Enclave using keys exchanged through ECIES. Note that the private key never leaves the enclave. [43]



Developers don't interact with the Secure Enclave directly. Instead, they can use the CryptoKit library in Swift or Security framework in Objective C if lower-level access is desired [44]. It is possible to specify **access control** for each generated key. Specification includes under which conditions the key should be accessible, such as only when the device is unlocked or if biometric authentication is required before each access. Private keys can be exported from the Secure Enclave in the form of an encrypted blob which can only be decrypted by the same enclave that generated it.

Persistence and additional access control are achieved through **Keychain**, a secret management system on Apple devices with a user-facing GUI application and APIs. It can be used to store generated keys or, in the case of keys generated by the enclave, their representations. It is implemented as an SQLite database (file on the filesystem), which is encrypted with two

keys. The table key encrypts the entire table (file), and then each record (row) is encrypted with a per-row key. Both keys are protected by the Secure Enclave, but the metadata key is also cached within the regular RAM, available to the OS, to speed up search queries. It is possible to configure ACLs for each record (key) in the database. The ACL entries can specify which specific applications have access to the key (e.g., using the Bundle ID, a unique application identifier), how and when the key can be accessed (e.g., biometrics), and other additional security policies. For example, a developer can require that after biometrics have changed, the can key no longer be accessed. This prevents the key even in the event of an attacker adding their own fingerprint or face scan. The Keychain database file cannot be accessed directly by applications. Instead, they have to call the Keychain API, which is handled by the `securityd` daemon (system service). Before allowing access to any database record (key), ACLs are verified.

Secure Enclave is used in both commercial and free and open-source applications. For example, a popular application, Secretive [45], for storing and managing SSH keys, uses it to store them. Every time a developer wants to push code to a source control system (git), they first need to authenticate using TouchID before the key itself is used for authentication. This is more secure than the common practice of leaving the SSH keys in plaintext on the filesystem. Commercial password managers also utilize the Secure Enclave, such as 1Password [46]. The master encryption key used to decrypt the password database is stored encrypted on disk until the user requests a password. The user is authenticated using TouchID, and the Secure Enclave decrypts the master encryption key, which is used to decrypt the database and retrieve the password. After a certain period of time, the decrypted master encryption key is securely deleted from memory.

In conclusion, while the Secure Enclave has many benefits, it also has some drawbacks. Unlike Intel SGX, Secure Enclave doesn't allow application developers to run custom code in the enclave. Pre-existing keys cannot be loaded, and the feature as a whole isn't supported on all Apple devices. Also, the selection of available algorithms is currently rather limited and might not suit all developers. The Secure Enclave also isn't invulnerable to attacks, although they can be tricky to exploit in the real world. In 2021, a security researcher was able to execute code on the T2 chip, given physical access to the device. While the attacker wouldn't be able to gain access to any secrets right away, it would be possible to run a keylogger on the device and write all secrets somewhere where the attacker could reach them. The issue affected computers released from 2018 to 2020. The researcher was awarded CVE-2021-30784 [47] for the finding, with Apple claiming to have mitigated the issue in the macOS Big Sur 11.5 OS update [48].

Despite these drawbacks, it appears that Secure Enclave enjoys wide adoption, especially on mobile devices. The possible reason behind that could be the availability of the standardized hardware (with enclave and biometrics) and the presence of standard native and high-level support in Swift and Objective C, without a need to resort to third-party frameworks.

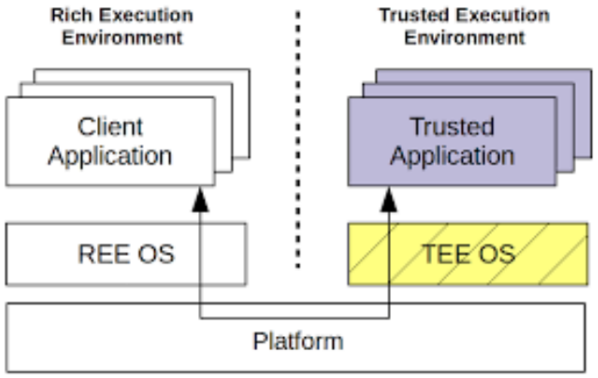
1.3.2 Trusted Execution Environments and Hardware Security Modules

Trusted Execution Environment (TEE) is a technology that allows running an isolated OS (Trusted Execution Environment) alongside a regular OS (Rich Execution Environment) on the same chip. It is typically found on ARM-based chips. The two can communicate through a layer called the secure monitor. An application running inside the TEE is called Trusted Application (TA) or a Trustlet. TAs usually provide a short list of services to reduce the attack surface. For example, a TA might offer a service to encrypt and decrypt data without giving out the encryption key. That way, even if the REE OS is compromised, the key inside the TEE remains secure. Typical usage is DRM, where a TA is responsible for decrypting the content and sending it directly to the GPU.

TEEs aren't supported by major x86 CPUs from Intel and AMD, so these features currently

aren't present on most desktop and server computers. However, as ARM desktop chips are becoming more popular, it is likely that this technology will be available and utilized soon.

Figure 1.11 Trusted Execution Environment - overview. The "platform" in the picture is commonly referred to as the secure monitor [49]



Hardware Security Module (HSM) is a specialized physical device designed to securely store and manage sensitive cryptographic keys and other digital assets. It can provide a highly secure environment for performing cryptographic operations and can be used for a variety of security-sensitive applications. A common use case is to hold TLS private keys in an enterprise environment.

Chapter 2

Sensitive Data in .NET

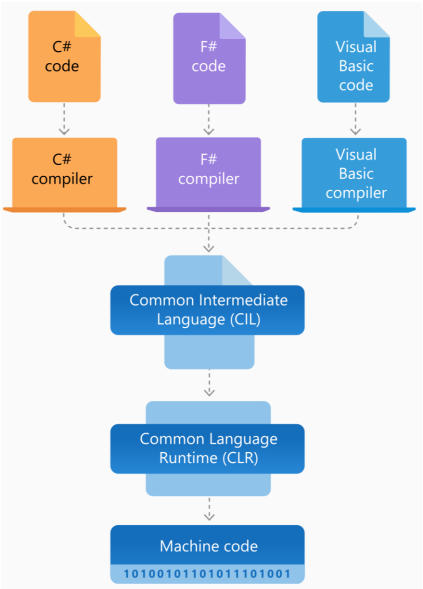
This chapter covers different memory protection techniques available in the .NET ecosystem and their effectiveness across different versions, architectures, and operating systems. The target language is C#.

2.1 C# and the .NET Ecosystem

C# is a multiplatform, type-safe, and object-oriented programming language with managed memory. It runs on a virtual execution system called Common Language Runtime (CLR), which is an implementation of the Common Language Infrastructure (CLI), an open standard.

.NET is an implementation of the CLR bundled together with a set of class libraries. Source code written in C# or other supported language (currently about 20 languages, e.g., F# or VisualBasic) gets compiled into the Intermediate Language (IL). When a program is run, CLR loads it and performs Just-in-Time (JIT) compilation into machine code, which is then executed on the CPU.

Figure 2.1 Overview of the .NET architecture [50]

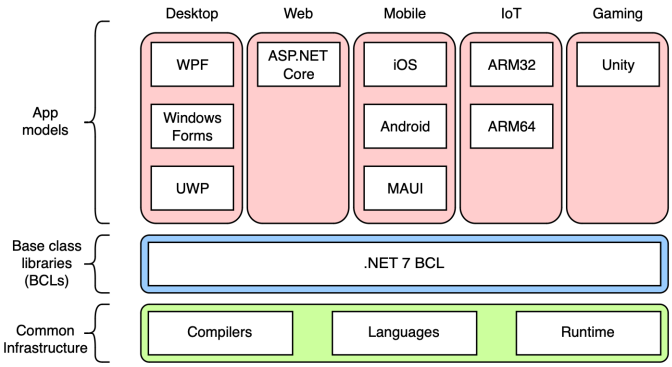


The original, Windows-only **.NET Framework** was introduced in 2002. It is not open source, but a reference source code is available. Although no new features are planned, it is still being supported and developed, and Microsoft pledged to continue to do so. The assessed version is 4.8.1 (current). In 2004, **Mono** was released by the Xamarin corporation. Mono is a cross-platform, open-source alternative to .NET Framework. It implements its own CLR and libraries, with the aim to maximize compatibility with .NET Framework. Windows, Linux, and macOS are supported. It is currently compatible with .NET 4.7, although some features are missing.

In 2016, Microsoft released **.NET Core**, a cross-platform (Windows, macOS, and Linux) and open-source alternative to .NET Framework. At release, it provided less functionality, but it slowly caught up and finally started adding new features unsupported by the original .NET Framework. In 2020, .NET Core was rebranded to simply **.NET**. It is currently the main edition of the framework, while .NET Framework isn't receiving new features.

Finally, on top of the runtime and BCLs, there are also application-specific frameworks (also called application models). While they are not technically part of the .NET (Framework), they are still considered a part of the ecosystem by Microsoft, mainly because of their tight integration. These extend the capabilities of .NET applications with a focus on particular use cases. For example, there are frameworks focused on GUI desktop/mobile applications or web applications.

Figure 2.2 Illustrative overview of different .NET ecosystem components, not comprehensive [51]



2.1.0.1 Desktop GUI Frameworks

There are many native desktop .NET GUI frameworks developed by Microsoft, many of them abandoned, and it can be difficult to navigate and differentiate between them. Microsoft, unfortunately, doesn't provide a comprehensive list of these frameworks, their use cases, or their level of support. On top of this, there are many community and 3rd party platform ports or alternatives that attempt to maintain compatibility with those original Microsoft frameworks. Covering all of them is impossible, so an attempt was made to include the most popular ones.

Windows Forms (WinForms) is a Windows-only GUI framework introduced in 2002, along with the original .NET Framework. It is currently supported by .NET as well, with the current version being 5.0. Because of its heavy dependence on Windows, it never became cross-platform. **Mono** provides a **cross-platform** alternative to Windows Forms that is currently compatible with Windows Forms version 2.0. However, UI elements don't look the same, and some aspects behave differently from Windows Forms. For a long time, this was the main way to develop C# cross-platform GUI applications.

Windows Presentation Foundation (WPF) was introduced in 2006 as an alternative to WinForms. It provides a more robust environment and capabilities, suitable for large applica-

tions, at the cost of increased complexity. Just like Windows Forms, it's Windows-only. There is currently no 1:1 cross-platform alternative like there is for Windows Forms, but community projects like Avalonia UI attempt to offer similar capabilities. There are, however, some 3rd commercial solutions that allow WPF applications to run on macOS on Linux. Both WinForms and WPF are still rather popular choices for developing Windows applications, with mature documentation and communities.

Universal Windows Platform (UWP), introduced in 2015, is a computing platform unifying various Microsoft platforms (Windows, Xbox, Hololens, ...) into one. Associated development toolkit allows developing GUI applications in .NET. One drawback of UWP is that developed applications can only be distributed through the Microsoft Store. It could be the reason behind it not seeming very popular.

The most recent addition to the list of native UI frameworks is **WinUI**. And again, it's Windows-only. Unlike UWP, it allows distributing the applications outside the Microsoft Store. There are some open-source projects like the Uno Platform that attempt to provide WinUI support on other platforms.

Finally, there are other frameworks like Xamarin.Forms and Multi-platform App UI (MAUI), which technically can run on some desktop platforms, but mainly target mobile devices.

2.1.0.2 Web Frameworks

ASP.NET is a web development Windows-only framework based on the .NET Framework, originally introduced in 2002. Alongside it, Microsoft introduced **ASP.NET Web Forms** as its simpler alternative. After releasing .NET Core, Microsoft also released the cross-platform **ASP.NET Core** based on it. ASP.NET was later renamed to **ASP.NET Framework 4.X**, and ASP.NET Core was rebranded as simply **ASP.NET** (although the "Core" name is still used in many contexts).

2.2 Memory Management

As was established earlier, C# is a programming language with a managed memory (so are other languages running on the .NET CLR). In order to protect the sensitive data in memory, it is best to keep them there for the shortest period of time possible, even more so when they are unprotected. Unfortunately, managed memory makes this a lot more complicated.

2.2.1 Garbage Collection

Data types in C# are essentially of three types: value types, reference types and references. Value types are simple data types like `int`, `bool`, or `double`. Reference types are classes, interfaces, delegates, objects, and strings (remember that array types are derived from `System.Array`, which is a class). References are, as the name suggests, references to objects defined using the ref keyword. Reference types are always stored on the **heap**; references are always stored on the **stack**. Value types can be stored on the stack or the heap, depending on the circumstances. Value-type variables that are part of a reference-type instance (e.g., a class property or an array item) are stored on the heap with the object itself. Value types declared locally are generally stored on the stack, with small exceptions, like outer variables of anonymous methods or iterator blocks. Developers can use boxing and unboxing, which transfers the variable to the heap and back. Finally, static variables are stored on the heap.

Understanding the storage location of data is important because the **garbage collector only manages the heap**. It is also important to note that while unlikely, the behavior described above is an **implementation detail and, as such, could potentially change at any time**. CLR doesn't make any guarantees.

The **garbage collector** (GC) [52] is a part of the CLR that manages the allocation and release of memory on the managed heap. The opposite of the managed heap is the native heap, which is managed directly by the operating system. While CLR makes no guarantees regarding when the memory is released, there are some general guidelines. Typically, it occurs if 1) the system is low on physical memory, 2) the combined size of all allocated objects is larger than a specific threshold (computed dynamically), or if the `GC.Collect` method is called. Because of points 1 and 2, garbage collection should be treated as unpredictable by the application developer.

When a garbage collection is triggered, the garbage collector releases the memory that's occupied by dead objects. The managed heap, just like any contiguous block of memory, can suffer from fragmentation. During garbage collection, GC can compact live objects by moving them together. This reduces fragmentation and shrinks the managed heap. While this is good for performance and memory usage, it means trouble from the perspective of sensitive data. **By moving the memory around the heap, GC can effectively create multiple copies of the same data**. Once a block of memory is managed by the GC, permanent deletion can never be guaranteed.

The heap itself is further divided into three regions called **generations**. The garbage collection frequency is highest for the generation 0 heap and lowest for the generation 2 heap. All objects are allocated in the gen. zero heap and move to upper levels as they survive garbage collections in their current generations. On top of this, there's also a special large object heap, sometimes referred to as the generation 3 heap, that's used to store large objects. These objects get allocated straight in that generation instead of moving from gen. 0 upwards. This and similar concepts create another layer of GC complexity and unpredictability for application developers.

There are two main ways to **take control** of a block of memory from the GC. First, it is possible to explicitly **instruct GC** to never move a certain object to a different place in memory. One way this can be done is with the `fixed` statement 2.1. However, that only guarantees no changes for that particular scope.

Code listing 2.1 Example of preventing the garbage collector from moving the contents of `secretArray` in memory

```
byte[] secretArray;
fixed (byte* pSecretArray = secretArray)
{
    // secretArray won't be moved by the GC
    // until the end of this block
}
```

Another possible way is to use the `GCHandle` structure, which provides a way to obtain a handle for a managed object. In particular, the `GCHandle.Alloc` method with an object and `GCHandleType.Pinned` as parameters instructs the GC to never move that object until the obtained handle is released by calling Free 2.2.

Code listing 2.2 Example of preventing the garbage collector from moving the contents of `secretArray` in memory

```
byte[] secretArray;
var gcHandle = GCHandle.Alloc(secretArray, GCHandleType.Pinned);

... // GC won't move secretArray in memory now

handle.Free(); // GC now can move the secretArray again
```

The second path for bypassing the GC is **storing data on the native heap** instead. It is riskier because it is possible to introduce memory leaks and security vulnerabilities, but it allows more control over the allocated memory. There are multiple ways to do it, with one of them being the `Marshal` class. It provides a collection of methods for allocating, manipulating, and releasing unmanaged memory in a platform-independent way. `Marshall.AllocHGlobal` method can be used to allocate unmanaged memory blocks of a desired size. It returns a handle (pointer) that has to be used to free the memory after usage 2.3. Internally, it uses the API of the OS to allocate the memory; for example, on Windows, it internally calls `LocalAlloc` (Win32 API).

Code listing 2.3 Example of allocating memory on the native (unmanaged) heap

```
// Allocate 100 bytes of unmanaged memory
IntPtr hglobal = Marshal.AllocHGlobal(100);

// Do something with the memory

// Release the memory
Marshal.FreeHGlobal(hglobal);
```

It is common to create wrappers around unmanaged resources to ensure their proper release. To make it easier, .NET offers the `SafeHandle` abstract class. It can be inherited from and used as a template.

`Marshal` class (and other .NET components) supports working with **BSTR**s (BaSic STRing) [53] in the native memory. BSTR is a simple structure containing length, the string itself, and a terminator character (`null`). There are dedicated methods for converting BSTRs into managed strings, byte arrays, unmanaged arrays of different types, and vice versa.

2.2.2 Secure Memory Deletion

In order to guarantee the **secure deletion** of an object in .NET, it must be **mutable and outside the reach of the garbage collector** (so pinned or allocated in the native heap). An example of an object that can never be reliably erased is the string because it is immutable - any change to a string just results in a new string being created. **Once data enters a managed string, it should be considered permanently exposed.**

Even when it's possible for an object to be securely deleted, there still could be problems. In theory, it is possible that a code erasing the memory could be **optimized away**. The **CLR doesn't make any guarantees about executing particular code, only about the end result**. However, this likely isn't a concern as of now. In the documentation, it is mentioned in multiple places that "while the C and C++ compilers have similar optimizations, no such optimizations are planned at this time for .NET." [54] However, as this could change in the future, it would be wise to ensure the memory is always deleted.

Optimizations can be disabled using the `MethodImplAttribute` attribute. If it's set to `MethodImplOptions.NoOptimization`, CLR should never optimize the contents of a given function away. It is recommended to bundle it `MethodImplOptions.NoInlining` as well, as shown in the implementation of the `CryptographicOperations.ZeroMemory` method 2.4. This is a dedicated method for securely erasing byte arrays. Unfortunately, it is only available in .NET (so not in .NET Framework or Mono).

Code listing 2.4 Implementation of the `CryptographicOperations.ZeroMemory` method .NET 7. Note the disabled CLR optimisations attributes.

```
[MethodImpl(MethodImplOptions.NoInlining |
                            MethodImplOptions.NoOptimization)]
public static void ZeroMemory(Span<byte> buffer)
{
 // NoOptimize to prevent the optimizer from deciding this
 // call is unnecessary
 // NoInlining to prevent the inliner from forgetting that
 // the method was no-optimize
 buffer.Clear();
}
```

The same concept applies to deleting memory allocated on the native heap, except there aren't any dedicated functions specifically for that. Of course, functions offered by the operating system, such as `ZeroSecureMemory` on Windows, can also be used.

If an object contains sensitive data (or unmanaged resources), it should implement the `IDisposable` interface. This allows the user of the object to erase it by calling the `Dispose` method or by using the using construct. It is common to call `Dispose` in `try/finally` block to ensure object deletion under all circumstances. Optionally, it is possible to also override a destructor or the `finalize` method (called by the GC) for the same purpose. However, **`IDisposable` should always be implemented**.

2.2.3 Memory Dumps

There are multiple ways in which the .NET application's memory can end up on disk. It is important to be aware of this risk and make sure that when that happens, only protected data can be found in the dump.

2.2.3.1 Crash Dumps

Whether or not crash dumps are created depends on the OS settings and .NET alone doesn't offer any way of preventing application memory from appearing in a crash dump. It is common to save crash dumps for debugging purposes and investigation, even in production environments. It is, therefore, critical to minimize the time the sensitive data is in memory.

2.2.3.2 Hibernation and Swap

The situation is similar with hibernation and swap files. Their creation and protection level depend on the policy of the OS. **.NET doesn't offer any way of preventing application memory from appearing in a hibernation or swap file**. However, there are some OS native functions that could be used, at least for the swap file. On Windows, there's the `VirtualLock` function that prevents a previously allocated block of memory from being swapped. The equivalent of this function for Linux and macOS is `mlock`. Preventing memory data from being stored in the hibernation file cannot be done on any OS.

2.2.4 Memory Management - Proof of Concept

As was established in this subchapter, understanding and controlling memory management is crucial to the protection of sensitive data. Unfortunately, while there is some research available for general GC behavior, there isn't much information regarding sensitive data exposure. The

information that the GC can move objects around the heap is widely available, but it's not clear whether most of the memory is overwritten by other objects as a consequence or there could be massive leaks. In order to continue with the assessment, this needs clarification. It is trivial to verify that GC doesn't interact with the native heap, so that doesn't need further analysis.

The goal of this experiment is to verify whether

- objects stored on the managed heap are at a real risk of being leaked, and if
- mechanisms that prevent GC from interacting are effective (e.g., pinning).

2.2.4.1 Setup and Results

An proof-of-concept application that conducts the following was created:

1. Allocate a number of identically-sized byte arrays on the managed heap.
2. Fill the arrays with a recognizable and repeating pattern.
3. Optional - pin some of these arrays in memory with `GCHandle.Alloc` and store the handle.
4. Collect addresses of the arrays.
5. Set some of the arrays (more precisely, the handle) to `null` to indicate to the GC it should collect them. Also, zero out their contents to prevent pollution of the results.
6. **Trigger garbage collection** with `GC.Collect` and `GC.WaitForPendingFinalizers`.
7. Collect the addresses of the arrays again.
8. Compile a report summarizing which arrays were moved in memory (and, optionally, how many were pinned).
9. Zero out all of the arrays (nothing should be in memory after this point unless the GC has moved objects around).
10. Wait for the process memory to be dumped so that it can be analyzed how many arrays have had their content leaked.

On top of these actions, additional constructs such as loops and arbitrary variables had to be added to prevent values and actions from being optimized away. They were discovered mainly through trial and error. Another issue was that `GC.Collect` does not necessarily force the garbage collector to shrink the heap (and move the objects), which is the main point of this test. Again, through trial and error, it was established that around 1/3 of arrays set to null is enough to trigger it under certain conditions. There either needs to be a large enough number of arrays, or the arrays need to be of a certain size. Table 2.3 summarizes these results.

Figure 2.3 The table shows what configuration of the proof-of-concept application resulted in the heap being shrunk and the data moved. There were only two situations. Either no arrays were moved, or all of them, introducing a large leak.

Array Count / Array Size	32	128	512	1024	4096
32	Stayed	Stayed	Stayed	Stayed	Stayed
128	Stayed	Stayed	Stayed	Stayed	Stayed
512	Stayed	Stayed	Stayed	*Moved*	*Moved*
1024	Stayed	Stayed	*Moved*	*Moved*	*Moved*
4096	*Moved*	*Moved*	*Moved*	*Moved*	*Moved*

It was discovered that when the GC moves the arrays, it **moves all of them to a new location**. This means that the **original memory stays behind mostly unchanged**. When the heap shrinkage happened, for all of the arrays, at least one copy of the data in memory was discovered after they were supposedly zeroed out. It was also established that **pinning the arrays worked well** and no leaks were identified for pinned arrays.

Code listing 2.5 Example output of the proof-of-concept application

```
Created 512 arrays of size 1024, pinned 8, set to null 170
[ENTER] to start garbage collection
GC moved 334/342 arrays, 8 arrays stayed in place, 8 pinned,
170 set to null
[ENTER] to clear arrays
Round finished. Dump process memory and observe if some of the data
is present
[ENTER] for another round, token=234
```

2.2.4.2 Conclusion

While these findings might be specific to this setup and not very scientific, they helped answer the key questions. **Using managed heap to store any sensitive data can result in massive leaks unless all objects are pinned**. It likely isn't straightforward or practical to demonstrate an attack on an application with such weakness because there's no guarantee when or if the GC decides to shrink the heap. However, it is reasonable to assume that the longer the application runs, the higher the probability.

The original tests were done on .NET 7 on Windows 11. Similar results were later obtained on both macOS and Linux, also with .NET 7 and Mono. Optimizations were enabled in all cases, just like with a production binary.

2.3 Memory Protection

.NET ecosystem offers different ways of protecting an application's memory. This subchapter analyses these available protection mechanisms and their implementation. The level of protection depends on the OS and framework version.

2.3.1 ProtectedData

`ProtectedData` is a class providing static methods for accessing the previously explored DPAPI functions, in particular, `CryptProtectData` and `CryptUnprotectData` (covered in chapter 1). `ProtectedData.Protect` and `ProtectedData.Unprotect` methods accept a byte array, additional entropy, and data protection scope - just like the original DPAPI functions. These functions are not as relevant to the topic of this thesis since their main goal is to **protect data at rest**. Nonetheless, they could still be used for that purpose, so it makes sense to cover them. Also, it is important to verify that Microsoft has used all necessary techniques to ensure no sensitive data is leaking during the execution of these functions.

The example usage [55] of these functions in the official documentation is a bit unfortunate. The example byte array that's being passed to the Protect method isn't erased from memory afterward. It is understandable in a simple use-case example, but let's assume that in the real world, a skilled developer would erase it as soon as possible. Furthermore, it also isn't pinned in memory.

Code listing 2.6 `ProtectedData.Protect` and `ProtectedData.Unprotect` method signatures

```
public static byte[] Protect (byte[] userData, byte[]? optionalEntropy,
    System.Security.Cryptography.DataProtectionScope scope);

public static byte[] Unprotect (byte[] encryptedData,
    byte[]? optionalEntropy,
    System.Security.Cryptography.DataProtectionScope scope);
```

2.3.1.1 Windows - .NET Framework

It seems that the .NET Framework (Windows-only) is working with the `CryptProtectData` and `CryptUnprotectData` functions correctly, at least when it comes to memory protection. Since `CryptProtectData` accepts input in the form of `CRYPTOAPI_BLOB` [56] and uses the same structure for output, it is important to pay attention to how they are allocated and released.

Both `userData` and `optionalEntropy` parameters are fixed in memory with `GCHandle.Alloc`. Pointers to the actual arrays are then used for constructing the `CRYPTOAPI_BLOB` structures. This doesn't create any additional copies of the original data, so it's not problematic. After the `CryptProtectData` function is called, it writes the data to a new unmanaged output buffer. Data from this buffer is then copied to a new byte array that's later returned. Finally, the output buffer is securely erased by calling a dedicated function, and all handles are released.

This looks a like a good example of how to work with sensitive data in .NET. **No sensitive data leaks were found in implementation**. The method makes sure that neither unprotected nor protected copies of the data stay in memory.

2.3.1.2 Windows - .NET

The implementation in the newer .NET is slightly different from the .NET Framework one. Instead of pinning the array in memory with `GCHandle.Alloc`, the fixed construct is used. Both `inputData` and `optionalEntropy` are fixed, and their pointers and used to create `CRYPTOAPI_BLOBs` (in here `DATA_BLOB`, but fundamentally the same structure). The temporary output buffer is again copied to the returned byte array and finally erased. Instead of calling a dedicated function to erase it, a simple for loop sets all of the elements to zero. In theory, this could be problematic because of it being optimized away if CLR were to ever do such optimizations, but Microsoft likely knows its CLR best.

Just like the .NET Framework implementation, this one doesn't seem to have any flaws when it comes to sensitive data in memory.

2.3.1.3 Linux and macOS

On both of these platforms, **.NET** throws a `PlatformNotSupportedException` exception when calling any `ProtectedData` method.

Mono implementation is entirely different from the previously inspected implementations. It doesn't use DPAPI, which is understandable, as that's not available on macOS or Linux. Instead, it uses its own implementation. The data is encrypted with a randomly generated AES key and IV. The AES key is then encrypted with a randomly generated **RSA key, which is stored in a static variable**. There are two RSA keys, one for the current user and one for the machine. The keys are also stored on disk, in the case of macOS in `/Users/<USERNAME>/.config/.mono/keypairs/<KEY ID>.xml`. The particular encryption scheme is not worth exploring in detail because since the keys are stored in a static variable, process memory alone is enough to decrypt the data.

This means that here, **Mono provides weak memory obfuscation rather than real protection**. Unlike with DPAPI, **access to the (decrypted) HDD image is enough to bypass the protection, and so is access to the process memory**.

2.3.2 Conclusion

In conclusion, on Windows, the ProtectedData class provides equivalent protection to DPAPI 2.4. This is because the methods are well implemented in both .NET and .NET Framework, so there are no leaks weakening the security. It would be helpful if the usage example showed the same level of security as the implementations themselves. Unfortunately, on Linux and macOS, it is either unsupported (.NET) or replaced with a custom, weaker implementation (Mono). Since Mono effectively provides a form of weak obfuscation but no real protection, process memory alone is everything an attacker needs.

Figure 2.4 What access level does an attacker needs to have to compromise data protected with `ProtectedData.Protect` on Windows (both .NET and .NET Framework)

Attacker Access Level	Read (Memory/File)	Execute Code
Process/User space	Protected	Vulnerable
Process crash dump	Protected	Vulnerable
Swap file	Protected	Vulnerable
Hibernation file	Vulnerable	Vulnerable
HDD (except hibernation)	Protected	Vulnerable
Root/Administrator	Vulnerable	Vulnerable
Kernel space	Vulnerable	Vulnerable
Hardware	Vulnerable	Vulnerable

2.3.3 ProtectedMemory

Like `ProtectedData`, `ProtectedMemory` is a class that provides static methods for accessing DPAPI functions through `ProtectedMemory.Protect` and `ProtectedMemory.Unprotect`. Both methods accept a byte array and a data protection scope, where the byte array's length must be a multiple of 16 - just like the original DPAPI `CryptProtectMemory` and `CryptUnprotectMemory` functions.

Surprisingly, the documentation doesn't mention the fact DPAPI is used or which particular functions. Just like with `ProtectedData`, the usage example in the documentation uses a byte array that's not pinned [57].

This is the main method for general memory protection that the .NET ecosystem offers. Unfortunately, **ProtectedData is only available in the .NET Framework**.

Code listing 2.7 `ProtectedMemory.Protect` and `ProtectedMemory.Unprotect` method signatures

```
public static void Protect (byte[] userData,
    System.Security.Cryptography.MemoryProtectionScope scope);

public static void Unprotect (byte[] encryptedData,
    System.Security.Cryptography.MemoryProtectionScope scope);
```

2.3.3.1 Windows

The implementation is a simple wrapper around the WinAPI `RtlEncryptMemory` (`SystemFunction040`) function, which it calls directly (`RtlDecryptMemory` for decryption). It doesn't create any copies of the plaintext or ciphertext, so there isn't a possibility of leakage (the data is encrypted in place).

In .NET the class doesn't exist at all.

2.3.3.2 Linux and macOS

Since the class doesn't exist in .NET, the only remaining possibility for macOS and Linux is Mono. Unfortunately, `PlatformNotSupportedException` is thrown when either of the methods is called.

2.3.3.3 Conclusion

This class could provide the main building block for memory protection in the .NET ecosystem, but unfortunately, it's only available for .NET Framework on Windows. The implementation itself doesn't seem to introduce any flaws, and based on the available research, the protection provided in that case should be equivalent to DPAPI (already summarized in the table 2.4). On other framework versions and platforms, the class either doesn't exist or throws an exception.

2.3.4 SecureString

`SecureString` [58] is a mutable class for storing text aimed at providing better sensitive data protection than a regular string. Its goals are to

- store the data in a memory outside the reach of GC,
- securely erase memory when possible,
- store the data only in an encrypted form, and
- prevent accidental exposure.

Possibly because of the functions it uses to manage the buffers, the maximum string length supported by `SecureString` is 65,536 characters. The following subsection investigates how exactly the `SecureString` is implemented on various platforms. Debugging and memory dumps were used to identify any potential sensitive memory leaks.

2.3.4.1 Windows - .NET Framework

The main buffer is always stored on the native heap, outside the reach of the GC. It is allocated by calling the WinAPI `SysAllocStringLen` [59] function, which returns a handle (pointer) to a newly allocated BSTR. Since it is always called with the initial string as null, the result is an empty BSTR of the desired length.

`SecureString` isn't managing this buffer by itself. Instead, it uses a dedicated `SafeBSTRHandle` class to wrap around the handle. Its original base class is the previously mentioned `SafeHandle`, but there are several levels of inheritance in between for additional capabilities. In the end, the class offers methods for allocation, secure disposal, obtaining and releasing pointers to the buffer, and reading and writing. This reduces the chance of making a mistake when working with unmanaged buffers and makes the code easier to audit. The only potentially problematic aspect is that for securely erasing memory, `ZeroMemory` **is called instead of** `SecureZeroMemory` (both WinAPI). However, as CLR currently isn't doing any optimizations that could result in the call being optimized away, it only might be a problem in the future. No sensitive data leaks were found in any of the methods the class provides.

When it comes to **protecting the buffer**, there are two methods - `ProtectMemory()` and `UnProtectMemory()`. Both are simple wrappers around `RtlEncryptMemory (SystemFunction040)` and `RtlDecryptMemory (SystemFunction041)`, respectively. The protection scope is set to `CRYPTPROTECTMEMORY_SAME_PROCESS`, so the key should be different for every process, and data in the `SecureString` should be lost once the process terminates. The methods only conduct a few rudimentary checks, like if the buffer length is a multiple of the `CryptProtectMemory` blocksize. Since it's passing the buffer handle directly and encryption is done in place, there isn't a possibility for a leak because **no data is being copied**. Both methods also make sure an exception is thrown in case anything goes wrong with the encryption process. This is important to prevent situations where it seems the encryption took place, but it actually didn't.

`SecureString` has two constructors, one parameter-less and one allowing to supply the initial value. The parameterless constructor only initializes the buffer with zero length through `SafeBSTRHandle` and verifies that DPAPI is supported by making a test call of `RtlEncryptMemory`. The other constructor accepts a pointer to a native char array. This means that a **managed array or a string cannot be directly used to initialize a** `SecureString`. As is explored later, this makes the object more secure while also making it harder to use. The `SecureString(char* value, int length)` constructor allocates a new buffer with `SafeBSTRHandle`, requests a pointer for it, copies the data directly from the old buffer, and encrypts them in place. The previously allocated pointer is then released. In the end, **neither of these constructors creates any additional copies of the data**, and allocated pointers are always released as soon as possible.

The need for **padding** is handled in a simple way. There are always two variables holding the size of the buffer: the first one stores the plaintext length (exposed as the `Length` property), and the second one stores the ciphertext length. The plaintext is always padded with zeroes to make sure the allocated length is a multiple of the `CryptProtectMemory` block size. No problems were identified with this scheme.

The `SecureString` correctly implements `IDisposable` to allow users of the object to securely dispose of it. It uses the analyzed verified `SafeBSTRHandle` methods to do so. After `Dispose` is called, neither ciphertext nor plaintext remains in memory. `SecureString` also offers the possibility to set it permanently as read-only by calling the `MakeReadOnly` method. It also isn't `Serializable` and doesn't override `ToString()`, so accidental exposure is less likely.

The string can be **modified** using four methods: `AppendChar`, `InsertAt`, `SetAt`, and `RemoveAt`. All of these methods work on an individual character level. The implementation of these methods is similar. It starts with verifying that `SecureString` hasn't been set as read-only. Then, `ApendChar` and `InsertAt` verify if there's enough capacity in the buffer for one more character. If there isn't, `SafeBSTRHandle` is used to allocate a new buffer of the target length, and the old buffer is copied into it and securely disposed of. The buffer is then decrypted, and operations themselves

(inserting, appending, or replacing a character) are done directly on it through a pointer requested from `SafeBSTRHandle`. These operations don't introduce any new copies of the data, so there are no leaks. Finally, the pointer is released, and the buffer is encrypted again.

Also, since these methods only accept char as a parameter, the risk of leakage is low (chars are allocated on the stack, which gets overwritten often and is outside of GC's control), and even then, it would be a single character.

Surprisingly, `SecureString` doesn't offer any public methods for obtaining the data inside. In order to get the decrypted value out of `SecureString`, the `Marshall` class must be used. It offers five methods 2.5 that create a new unmanaged buffer, copy the decrypted data there and return a handle (pointer) to it. The implementation of these methods is rather simple - they are only calling dedicated `SecureString`'s internal methods. For example, `Marshal.SecureStringToBSTR` calls `SecureString.ToBSTR`. It then allocates a new buffer with `SysAllocStringLen`, decrypts the original buffer, obtains a pointer to it through `SafeBSTRHandle`, copies its contents to the new buffer, encrypts the original buffer back, releases the pointer, and finally returns a pointer to the newly allocated buffer. Other methods `Marshall` class offers differ mainly in the encoding of the resulting string, with their implementations staying similar. The main difference is that instead of allocating the new buffer with `SysAllocStringLen`, they use either `Marshall.AllocHGlobal` or `Marshal.AllocCoTaskMem` (both allocating unmanaged memory).

Figure 2.5 Methods for copying the buffer out of the SecureString into a new buffer, and their corresponding methods for erasing it [58]

Allocation and Conversion Method	Corresponding Zero and Free Method
`Marshal.SecureStringToBSTR`	`Marshal.ZeroFreeBSTR`
`Marshal.SecureStringToCoTaskMemAnsi`	`Marshal.ZeroFreeCoTaskMemAnsi`
`Marshal.SecureStringToCoTaskMemUnicode`	`Marshal.ZeroFreeCoTaskMemUnicode`
`Marshal.SecureStringToGlobalAllocAnsi`	`Marshal.ZeroFreeGlobalAllocAnsi`
`Marshal.SecureStringToGlobalAllocUnicode`	`Marshal.ZeroFreeGlobalAllocUnicode`

It is the **responsibility of the caller to ensure the data gets securely deleted** after use. For this, the `Marshall` class offers a corresponding method for each of five of the previously introduced ones 2.5. Implementation of these methods differs mainly in the size of the memory being erased to account for differences in encoding.

While this architecture might seem complicated, it is arguably a **good way to prevent accidental exposure**. By moving possible exposure methods outside of the `SecureString` into the `Marshall` class, developers are forced to only expose the data in a deliberate way.

Overall, `SecureString` implementation in the .NET Framework proved to be resistant to data leaks and thoughtfully designed.

2.3.4.2 Windows - .NET

The implementation of `SecureString` in .NET on Windows is quite similar to the .NET Framework one, but there are differences. Since it has to be cross-platform, **WinAPI functions aren't used** (directly) anymore (except for the protection). Instead of `SysAllocStringLen`, `Marshal.AllocHGlobal` is used for the allocation of the unmanaged buffer, along with the corresponding `Marshal.FreeHGlobal`. The wrapper around it has been renamed from `SafeBSTRHandle` to `UnmanagedBuffer`, but its functionality remained identical.

Instead of using the WinAPI `ZeroMemory` function, it uses `Span<T>.Clear()`. `Span<T>` [60] is a mutable object representing a region of memory. It is only available in .NET and is **guaranteed to be stored entirely on the stack**. This ensures that it's outside the reach of GC. `Span<T>` is used extensively in this implementation. Apart from overwriting buffers, it replaces the char

operations of the `SecureString` modification methods (e.g., `SetAt`) and internal methods used by the `Marshall` class. Apart from the hypothetical issue of it being optimized away, no issues were identified (no data leaking).

Protection is nearly identical to .NET Framework, with one small change - `CryptProtectMemory` is used instead of `RtlEncryptMemory` (and analogically for decryption). It is not clear why this is, but as was established in the first chapter, the security provided by these functions is equivalent.

On Windows, this implementation proved at least **as secure as the .NET Framework one**.

2.3.4.3 Linux and macOS

Unfortunately, on Linux and macOS, problems manifest. The **Mono** implementation uses a **regular managed byte array** as the main buffer. Furthermore, **no encryption is used at all**, so the data is always stored in plaintext.

The implementation in **.NET** is more robust than in Mono, but unfortunately, it also isn't using encryption to protect the data.

Code listing 2.8 Implementation of the SecureString memory protection ("encryption") in .NET on macOS and Linux

```
private void ProtectMemory()
{
    _encrypted = true;
}

private void UnprotectMemory()
{
    _encrypted = false;
}
```

Both of these cases have fundamental issues, but .NET provides better protection when it comes to disposing of the object securely and preventing accidental exposure. After all, it's the same implementation as on Windows; just the DPAPI part is missing.

2.3.4.4 Optimal Usage

In order to use `SecureString` effectively, strict precautions must be taken. `SecureString` should ideally be created empty and then built by adding one character at a time 2.9. If that's not possible, it can be initialized from an unmanaged buffer. Once the instance is not needed anymore, it should be disposed of immediately by calling `Dispose`, preferably in a `try/finally` block (in case an exception is thrown, the data should still be securely deleted).

Code listing 2.9 Example of correctly loading data into `SecureString` from the official documentation [58]. Note the `try/finally` block and that `SecureString` is being immediately disposed of once it's not needed.

```
// Instantiate the secure string.
SecureString securePwd = new SecureString();
ConsoleKeyInfo key;

Console.Write("Enter␣password:␣");
do {
   key = Console.ReadKey(true);

   // Ignore any key out of range.
   if (((int) key.Key) >= 65 && ((int) key.Key <= 90)) {
      // Append the character to the password.
      securePwd.AppendChar(key.KeyChar);
      Console.Write("*");
   }
// Exit if Enter key is pressed.
} while (key.Key != ConsoleKey.Enter);
Console.WriteLine();

try {
    Process.Start("Notepad.exe", "MyUser", securePwd, "MYDOMAIN");
}
catch (Win32Exception e) {
    Console.WriteLine(e.Message);
}
finally {
   securePwd.Dispose();
}
```

Sensitive data should never enter a managed string. Ideally, once the data enters `SecureString`, it should never leave it. Only if it's absolutely necessary the `Marshall` class can be used to temporarily obtain an unmanaged buffer 2.10. This buffer should then be disposed of as soon as possible by calling the appropriate function (depending on which function was used to obtain the buffer).

Code listing 2.10 Example of correctly obtaining a pointer to `SecureString`'s decrypted unmanaged and securely disposing of it (official documentation [61], shortened)

```
...
SecureString password = ... // passed as a parameter

// Marshal the SecureString to unmanaged memory.
passwordPtr = Marshal.SecureStringToGlobalAllocUnicode(password);

// Use the decrypted buffer
returnValue = LogonUser(userName, domainName, passwordPtr,
                    LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT,
                    ref tokenHandle);
...
// Zero-out and free the unmanaged string reference.
Marshal.ZeroFreeGlobalAllocUnicode(passwordPtr);
...
```

2.3.4.5 Use Throughout .NET Ecosystem and Windows

`SecureString` **is present throughout the .NET ecosystem**, including GUI frameworks like WPF. In .NET and .NET Framework, `SecureString` is used by the `Process`, `NetworkCredential`, `SqlCredential`, `EventLogSession`, `CspParameters`, and `X509Certificate` classes, to name a few. It is commonly used to pass a password to a method or store it as a class property. WPF also uses it inside the `PasswordBox` control.

`SecureString` is also utilized by **PowerShell** [62]. It relies on it for handling credential input from the user (if `SecureString` is used, the password automatically isn't echoed). Powershell extends the implementation with additional functionalities, like allowing persistence. For example, `SecureString` can be exported as a DPAPI blob.

2.3.4.6 Microsoft's Stance

Despite `SecureString`'s extensive use, **Microsoft doesn't recommend it for new development** [58]. However, no alternative is being provided. Instead, the recommendation is to avoid using credentials entirely and use other means of authentication, such as certificates, or to "use an opaque handle to credentials that are stored outside of the process." How those credentials outside of the process should be protected is not specified. It also isn't established how or if to protect other types of sensitive data, e.g., credit card numbers.

Despite this recommendation, `SecureString` isn't set as obsolete, and given its wide usage, it's unlikely it will be in the near future. This leads to a confusing situation where the current solution is slightly discouraged without any real alternative. Many of the previously mentioned objects in .NET ecosystem can't even be used without providing `SecureString`, for example, `Process.Start` when starting a process as another user.

2.3.4.7 Conclusion

`SecureString` **on Windows is a well-designed and well-implemented class**. Both .NET Framework and .NET implementations could serve as a good example of how to work with sensitive data in C# and what to pay attention to. Because there are no implementation flaws, such as sensitive data leaks, the security level provided is identical to the security of DPAPI (again, table 2.4). `SecureString` uses techniques to prevent accidental exposure and forces the developer to be deliberate about exposing its contents. This aspect, unfortunately, also makes it difficult to use (explored later). However, there's only so much it can do. There's still a lot of pressure on the developer to use the class correctly and to make sure not to expose the data anywhere else in the application. Still, on its own, `SecureString` on Windows appears to be a sound solution, given the circumstances.

Unfortunately, on **other platforms**, it becomes **nearly useless**. .NET implementation could still theoretically be used at least to prevent accidental exposure and to minimize the time data is in memory, but the main feature is missing. Mono's implementation doesn't even provide that. Since no protection is used in either case, the data will be in plaintext.

The arguments made by Microsoft do make sense. In Windows or any other operating system, `SecureString` doesn't exist as a concept. Unless `SecureString` is supported by every library and every function an application uses, developers are going to have a hard time using it to its fullest potential.

2.4 UI and Proof-of-Concept Applications

While it is essential to assess the security of the core components like the `SecureString`, in the real world, an application comprises many different modules. The key concept is that an application might be less secure than the sum of its parts.

This subchapter aims to contrast a simple application without any previously introduced protection measures with an application that utilizes the best of what the .NET ecosystem has to offer in this area. The point isn't to build a superior solution from scratch but rather to use components and approaches that are already available - like a regular application developer might do.

Unfortunately, in this area, there is little to be gained by exploring the .NET ecosystem on macOS and Linux any further. As was established earlier, protections are either non-existent or trivial to bypass.

2.4.1 Objective

The goal is to create a **simple application for encrypting and decrypting files with a symmetric key**. This should be sufficient to demonstrate operations with sensitive data such as passwords, keys, and the sensitive files themselves. The application should ask the user for a password, then derive an encryption key from it and use it to encrypt the file. It should also be possible to remember the password until the program is shut down.

The target platform is **Windows with .NET**. While .NET Framework currently supports one memory protection mechanism (`ProtectedMemory`), it is not useful for this experiment. Note that the goal isn't a fully-fledged production-ready application for encrypting files. The encryption of files merely serves as a placeholder for operations that can leak sensitive data.

2.4.2 Naive WinForms Application

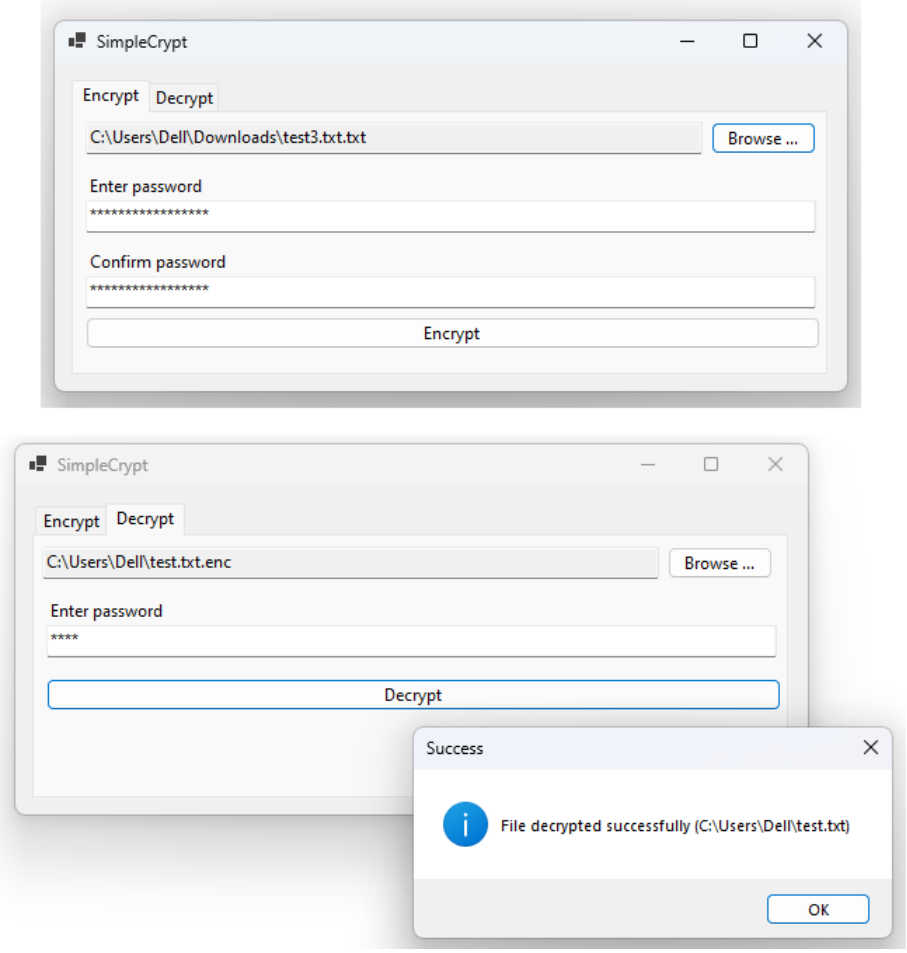
SimpleCrypt WinForms is an application based on Windows Forms. As such, it can use whatever elements that the framework makes available. Unfortunately, **Windows Forms doesn't have a dedicated secure UI element for password entry**. However, in multiple official tutorials, Microsoft recommends using a regular TextBox instead [63]. It has a `PasswordChar` property, which can be used to hide the characters in the textbox when the user is typing a password. Since nothing better is available, the application uses this method. The application allows the user to select a file to be either decrypted or encrypted with an `OpenFileDialog`. The resulting encrypted files are saved to the same location as the source files with ".enc" extension. There's no need to specifically provide a UI element for persisting the password, as that is done implicitly. The entered password is used to derive a key with PBKDF2, which is used to encrypt or decrypt the file with AES GCM. The code isn't trying to protect any information in memory and completely relies on the GC.

2.4.2.1 Security issues

As expected, the application **exposes sensitive data on many occasions**. Because TextBox only allows retrieval of the text as a string, this alone is enough to compromise security. The password is almost guaranteed to show up in a memory dump. Further manipulation with the password ensures that there will be multiple instances of it.

However, it's not just the password that's leaking. It is rare to see byte arrays with sensitive data being overwritten in C# examples online (including the official documentation), so this application isn't doing it too. This means that **memory dumps also contain the derived key, plaintext, ciphertext, and all of the intermediate values like the nonce**.

Figure 2.6 SimpleCrypt WinForms, a proof-of-concept application



Code listing 2.11 Encrypting a file in SimpleCrypt WinForms (higher level). Note that none of the byte arrays are being overwritten or pinned.

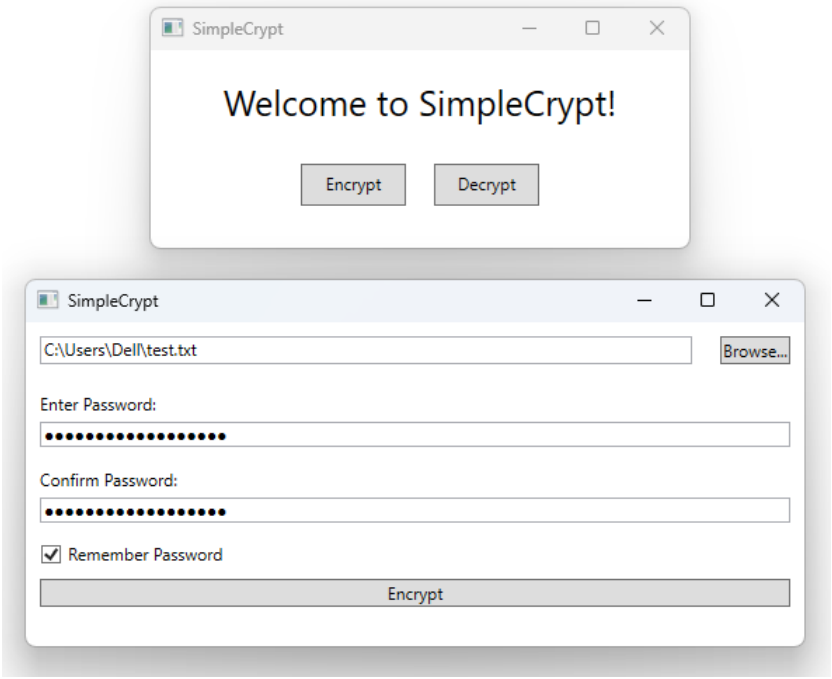
```
public static string Encrypt(string path, string password)
{
    byte[] key = CryptoHelper.DeriveKey(password);

    byte[] plaintext = File.ReadAllBytes(path);
    byte[] ciphertext = AesGcmHelper.Encrypt(plaintext, key);

    string encryptedFilePath = Path.GetFullPath(path) + ".enc";
    File.WriteAllBytes(encryptedFilePath, ciphertext);

    return encryptedFilePath;
}
```

Overall, the application provides a good example of how online documentation examples might be used to put together a problematic piece of code.

2.4.3 Protected WPF Application

This counter-example application aims at protecting all of the data it can. In this case, it is easier because WPF offers `PasswordBox`, a more secure alternative to `TextBox` which uses a `SecureString` and builds the password from characters. The functionality of the application is identical to the vulnerable one - encryption and decryption of files with AES GCM, where the key is derived with PBKDF2.

Figure 2.7 SimpleCrypt WPF, a proof-of-concept application



While using `SecureString` has security benefits, it creates problems for the developer. Most of the methods available with string are intentionally missing, and one has to be careful not to introduce a vulnerability by improper manipulation. As a result, the code is longer and more convoluted. First, it is important to call `Dispose` whenever the `SecureString` isn't needed. Since the application uses `SecuresString` to temporarily store the password when the user requests it, this alone warrants a couple of if statements and method calls in various places. Furthermore, in order to compare that the two passwords match, a dedicated method needs to be written (`SecureStrings` cannot be compared to other `SecureString` by default). Luckily, Microsoft has an internal version of this method in the .NET Framework source code 2.12, which can be adapted. This method is well-written and doesn't introduce any leaks.

Code listing 2.12 Securely comparing two SecureStrings (internal method in the .NET source code) [64]

```
public unsafe static bool Compare(SecureString secureString1,
    SecureString secureString2)
{
    ...
    try
    {
        bstr1 = Marshal.SecureStringToBSTR(secureString1);
        bstr2 = Marshal.SecureStringToBSTR(secureString2);
        result = true;
        for (int i = 0; i < secureString1.Length; i++)
        {
            if (*((char*)bstr1 + i) != *((char*)bstr2 + i))
            {
                result = false;
                break;
            }
        }
    }
    finally
    {
        if (bstr1 != IntPtr.Zero)
            Marshal.ZeroFreeBSTR(bstr1);
        if (bstr2 != IntPtr.Zero)
            Marshal.ZeroFreeBSTR(bstr2);
    }
    return result;
}
```

Another issue is that the native .NET methods for key derivation don't support `SecureString`. This means that a dedicated wrapper method needs to be written. Fortunately, there is a good community [65] implementation available. Essentially, `SecureString` needs to be loaded into a pinned managed byte array (the key derivation method doesn't accept unmanaged buffers), and the temporary array has to be cleared. Finally, there's the issue of all those managed byte arrays that hold the derived key, nonce, ciphertext, and plaintext. As was established earlier, unmanaged byte arrays pose a real security risk, even when they are cleared. Unfortunately, there isn't an easy way around this. All byte arrays holding sensitive data have to be pinned, cleared, and then the handle has to be released. This process makes the code harder to read and also places a burden on the developer 2.13. For these byte arrays, the responsibility put on the developer is similar to lower-level languages with unmanaged memory.

With all of these measures implemented, the application now doesn't leak any of the sensitive information (tested with a debugger and multiple memory dumps).

2.4.4 Other UI Frameworks

Surprisingly, newer .NET UI Frameworks, such as UWP, **don't offer an alternative to the WPF `PasswordBox`**. For example, UWP `PasswordBox` [66] uses a regular string to work with passwords.

2.4.5 Conclusion

This experiment demonstrated that if a .NET application developer doesn't make a considerable

effort, the application is likely to end up insecure. And even if the developer is aware of the risks, it can be trivial to make a mistake. The lack of UI components and libraries supporting secure strings (like crypto libraries) requires additional code to be developed.

Code listing 2.13 Encrypting a file in SimpleCrypt WPF (higher level). Note that every byte arrays has to be pinned, cleaned and the handle released.

```
public static string Encrypt(string path, SecureString password)
{
    byte[] key = CryptoHelper.DeriveKey(password);
    GCHandle keyGCHandle = GCHandle.Alloc(key, GCHandleType.Pinned);
    byte[] plaintext = File.ReadAllBytes(path);
    GCHandle plaintextGCHandle = GCHandle.Alloc(plaintext,
                                                GCHandleType.Pinned);

    GCHandle ciphertextGCHandle = new GCHandle();
    byte[] ciphertext = null;

    try
    {
        ciphertext = AesGcmHelper.Encrypt(plaintext, key,
                                          out ciphertextGCHandle);

        string encryptedFilePath = Path.GetFullPath(path) + ".enc";
        File.WriteAllBytes(encryptedFilePath, ciphertext);

        return encryptedFilePath;
    }
    finally
    {
        CryptoHelper.ZeroByteArray(key);
        CryptoHelper.ZeroByteArray(plaintext);
        if (ciphertext != null)
            CryptoHelper.ZeroByteArray(ciphertext);

        keyGCHandle.Free();
        plaintextGCHandle.Free();
        ciphertextGCHandle.Free();
    }
}
```

2.5 Conclusion

Memory protection in the .NET ecosystem is a complicated and confusing topic. In order to effectively protect an application, it is important to understand how the memory is managed and what mechanisms are available to take control back. While the fact that the GC can move the memory around the heap didn't seem significant, it was demonstrated that large leaks could happen.

Depending on the version and target OS, the .NET ecosystem offers more or less powerful memory protection functionality. It is surprising that the best-equipped combination for memory protection is the .NET Framework and WPF instead of their more recent counterparts.

Unfortunately, the .NET ecosystem isn't utilizing any of the advanced hardware protection

mechanisms introduced in the first chapter, like Intel SGX, TPMs, Trusted Execution Environments, and the Secure Enclave. The best level of security is provided by DPAPI, which is still vulnerable to code execution on any level, and full RAM dump alone is enough to compromise it. Protection outside Windows is essentially non-existent, forcing the developers to look for alternatives.

On Windows and when evaluated in an isolated environment, `SecureString` is a well-designed class that provides a reasonable level of protection. However, in order for it to become effective, it has to be supported by other parts of the framework, including the UI elements. In its current state, developers are required to go above and beyond to use it with other parts of .NET, let alone 3rd party frameworks.

Furthermore, an improvement in interoperability seems rather unlikely. Microsoft doesn't recommend `SecureString` for new development, and the `ProtectedMemory` class is still only available in the .NET Framework. Part of the problem is likely on the operating system level because constructs like the `SecureString` simply don't exist there.

A portion of problems could also be attributed to the design of high-level languages in general. If a developer wants to protect an application's memory, it will likely mean giving up on many features that make high-level languages easy to use - a big part of their appeal. As was demonstrated in the proof-of-concept application, improving memory protection beyond a certain level results in managing various handles manually, similarly to memory management in C.

Chapter 3

Real-World Usage - KeePass Password Manager

KeePass [67] is a free and open-source password manager by Dominik Reichl, first released in 2003. It is written in C# and supports Windows, macOS, and Linux. On Windows, it runs on .NET Framework; on macOS and Linux, it runs on Mono. The current version targets .NET Framework v4.6.2, so that's what it was tested with. It was audited multiple times [68], including by government agencies, and also had a financed bug bounty program. It is, therefore, less likely for it to contain major vulnerabilities. The analyzed version of KeePass is 2.53.1 with default settings.

On its website, KeePass author claims that *"while KeePass is running, sensitive data is stored encryptedly in the process memory. This means that even if you would dump the KeePass process memory to disk, you could not find any sensitive data. For performance reasons, the process memory protection only applies to sensitive data; sensitive data here includes for instance the master key and entry passwords, but not user names, notes and file attachments ... Furthermore, KeePass erases all security-critical memory (if possible) when it is not needed anymore, i.e. it overwrites these memory areas before releasing them."* [69]

The project is structured into two parts. KeePassLib provides helper and utility classes, and KeePass is the main application. The project utilizes quite a few preprocessor directives, such as KeePassLibSD (for compiling for PocketPC/Windows Mobile) or KeePassUAP (experimental support for Universal Windows Platform). Unless they were relevant to the target platforms, they are omitted from the analysis and code snippets below for simplicity.

3.0.0.1 SecureString Usage

KeePass doesn't use SecureString. This is shortly explained in the source code, as well as multiple times on KeePass' forums by the author himself. Reasons for not using it include a lack of flexibility and insufficient protection in the Mono environment, which corresponds to the findings in the previous chapter. Instead, the author developed **ProtectedBinary** and **ProtectedString** classes as a replacement, which can be found in the KeePassLib project.

3.1 ProtectedBinary

`ProtectedBinary` is an immutable object that, from the user's perspective only allows initialization and reading of the decrypted data. It is essentially an encrypted byte array. It uses a regular **managed** `byte[]` type for data storage. The array, `m_pbData`, **isn't pinned** in memory, so as was

established earlier, a garbage collector might move it around in a heap, leaving traces of data behind. When it comes to deleting the byte array, it is done correctly. There's a dedicated utility function with disabled optimizations 3.1, which is used every time an array needs to be deleted. This is likely the best possible way to deal with this since the application doesn't support .NET Core and, later, so `CryptographicOperations.ZeroMemory` is not available.

Code listing 3.1 A utility method used throughout KeePass to securely delete memory. Note that optimizations are disabled to ensure that no call gets optimized away.

```
private const MethodImplOptions MioNoOptimize =
(MethodImplOptions.NoOptimization | MethodImplOptions.NoInlining);
...
[MethodImpl(MioNoOptimize)]
public static void ZeroByteArray(byte[] pbArray)
{
   if(pbArray == null) { Debug.Assert(false); return; }

   Array.Clear(pbArray, 0, pbArray.Length);
}
```

The class has four constructors, out of which one is parameter-less for creating empty objects, and three have a mandatory `bool` parameter `bEnableProtection`. As the name suggests, it is used to indicate whether the data should be stored encrypted or not. This is a debatable design choice, as it's easy to create a `ProtectedBinary` that uses no encryption at all by mistake. However, since this application is mainly developed by one person, it likely isn't an issue, as it would be in a larger team. Individual use cases in the code likely don't pose a security risk because the unencrypted version is only used for interfacing with other password formats (exporting a password database) and serialization - a situation where this might be desirable. Once the object gets created as protected or unprotected, it cannot be changed.

The protected version is used more often; most use cases cover the protection of cryptographic secrets like keys or entropy.

Code listing 3.2 Typical usage example of `ProtectedBinary`. A byte array is filled with sensitive data, `ProtectedBinary` is constructed, and the original unprotected array is deleted right after.

```
byte[] pbNewPool = shaPool.ComputeHash(pbCmp);
m_pbEntropyPool = new ProtectedBinary(true, pbNewPool);
MemUtil.ZeroByteArray(pbNewPool);
```

`ProtectedBinary` doesn't implement `IDisposable`, a destructor, `Clear`, or any other method for securely erasing it. This means that there is no way to clear sensitive data from memory once they are not needed.

The class is correctly using the lock statement, together with Interlocked features, to stay thread safe.

3.1.1 Encryption

The class supports four different protection modes, which it titles as follows: None (no protection), `ProtectedMemory`, `ChaCha20`, and `ExtCrypt`. The last one, `ExcCrypt`, allows a KeePass plugin to supply its own encryption method through a delegate. This option was omitted from

the analysis. The encryption is always done in place, in the `m_pbData` property, the byte array dedicated for main storage.

On **Windows**, the memory is encrypted by calling `ProtectedMemory`.Protect with `MemoryProtectionScope.SameProcess` flag. This might be the explanation behind the choice of `byte[]` as the core type, as well as the block size. The need for padding is solved in a simple way. Block size is hardcoded to 16, and if the input isn't a multiple of 16, it gets padded with zeroes. The real size of the input is kept in the `m_uDataLen` property, so this is effectively hidden from the user of the object. This mechanism doesn't introduce any security flaws. It is clear that on Windows, dumping the process memory alone won't result in the ability to decrypt encrypted data because the encryption key isn't stored there.

The situation gets worse on **Linux and macOS**, where the class falls back to its custom encryption mechanism using ChaCha20. It starts by generating a random 256-bit key with a custom function, `Protectedbinary.GetRandom32`. This key is then stored as a static class property `g_pbKey32` (so all `ProtectedBinary` instances are encrypted with it). The IV is a long counter also stored as a static class property that gets updated only when a new object is initialized. While this sounds like a potential security issue because reusing IV in a stream cipher can lead to exposing the encryption key, keep in mind that the object is immutable. While it can be encrypted and decrypted multiple times throughout its lifetime, the data being encrypted as well as the IV, stay identical until disposal.

The custom function for random generation, `Protectedbinary.GetRandom32`, starts by generating 32 random bytes with `RNGCryptoServiceProvider`. While this class is currently obsolete [70], this fact doesn't seem to have any security implications. However, it is recommended to use `RNGCryptoServiceProvider` instead. Another 28 bytes of entropy are then obtained by calling `Guid.NewGuid()` and saving tick count and the current date. The quality of this additional entropy is objectively very poor, as neither of these values is anywhere near random. Using `Guid.NewGuid()`, for this very purpose, is clearly discouraged by Microsoft [71]. System time and tick count are even worse. These 60 bytes are then hashed with SHA256, and the result is returned.

The way these two entropy sources are put together also has some problems 3.3. They are both in separate try/catch blocks that do nothing if any exception is thrown. So if `RNGCryptoServiceProvider` throws an exception during generation, such as when the system is low on memory, the program will resort to using only the low-quality entropy source without any user alert or log entry of this happening.

Code listing 3.3 Combining entropy sources in `RNGCryptoServiceProvider` (simplified). Note that if an exception is thrown in the first try block, it will be ignored and inferior entropy source is used instead.

```
byte[] pbAll = new byte[32 + 16 + 8 + 4];
int i = 0;
try
{
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
    byte[] pb = new byte[32];
    rng.GetBytes(pb);

    Array.Copy(pb, 0, pbAll, i, 32);
    i += 32;
    ...
}
catch(Exception) { Debug.Assert(false); }

try // In case RNGCryptoServiceProvider doesn't work properly
{
    byte[] pb = Guid.NewGuid().ToByteArray();
    Array.Copy(pb, 0, pbAll, i, 16);
    i += 16;

    MemUtil.Int64ToBytesEx(DateTime.UtcNow.ToBinary(), pbAll, i);
    i += 8;

    MemUtil.Int32ToBytesEx(Environment.TickCount, pbAll, i);
    i += 4;
}
catch(Exception) { Debug.Assert(false); }
```

Furthermore, the ChaCha20 function used for the encryption is a custom one. It is unknown why the author chose to do his own implementation of this function, despite the wide availability of libraries that support it. Evaluating the particular implementation is out of the scope of this thesis, but it should be noted that this practice is commonly discouraged.

However, all of the potential issues identified have very little impact in reality. Since the encryption key is stored alongside the encrypted data without any level of isolation, this protection is really **more obfuscation than encryption**. Indeed, an attacker with access to the process memory dump doesn't need any other resource to obtain the plaintext. It might require some work because the encryption key will likely be stored further away from the ciphertext because it's a static field. Simple process memory analysis on Linux (Mono) revealed that the IV is stored next to the ciphertext, and since the IV is sequential from 0, a clear pattern emerges of random N bytes, followed by a low number stored as an 8-byte integer. From there, it's possible to find the original handle (pointer) in the `ProtectedBinary` object's memory, and that leads directly to the address of the static encryption key byte array.

3.1.2 Usage

The object is used to protect all kinds of cryptographic secrets, even the master key itself. When it comes to accidental exposure, some areas are good, like the object not overloading methods like ToString or not being serializable. This should be effective in preventing mistakes like writing this object to logs or a file. However, the previously mentioned ability to construct objects easily without any memory protection could certainly cause problems.

Two cases were identified where the object isn't used effectively because the original data source isn't immediately cleared from memory. These are likely rather minor findings:

- `KcpCustomKey.cs:58`
  Array `pbRaw` containing the key hash isn't cleared from memory before being out of scope.
- `KcpCustomKeyBinaryDataUtil.cs:150`
  Array `pbModData` containing sensitive binary data isn't cleared from memory before being out of scope.

3.2 XorredBuffer

`XorredBuffer` is a simple class that stores ciphertext and a corresponding XOR pad of equal length, both as byte arrays. The constructor accepts two parameters, an XOR pad and a plaintext, which is transformed into ciphertext and stored as a class property. `ReadPlainText` method can then be used to decrypt the ciphertext and return the result as a byte array. It implements `IDisposable`, which correctly erases both of its arrays. Alone, it essentially provides simple obfuscation. Both `ProtectedBinary` and `ProtectedString` accept it in their constructors. It is part of a scheme discussed later.

3.3 ProtectedString

`ProtectedString`'s purpose is to provide an alternative to the .NET `SecureString` object. It doesn't do any encryption itself but uses the `ProtectedBinary` class to store the data instead. There are a lot of architectural similarities between the two classes. Similar to `ProtectedBinary`, `ProtectedString` has four constructors, with the default one being empty with protection disabled and the rest allowing to either enable or disable protection via a `bool` parameter. It is also possible to obtain an empty `ProtectedString` with protection enabled through a static property `EmptyEx`. These design choices were already discussed for `ProtectedBinary`. However, one of the constructors, which accepts a string, has this problem even more pronounced 3.4. Just like the others, its first parameter is `bEnableProtection`. However, in this case, no matter the value, it is always ignored! On the one hand, this makes sense. After all, once data enters memory as a managed string, there's nothing that can be done to guarantee it will be deleted. On the other hand, why should this be an available constructor of a `ProtectedString`? If an engineer calling this function doesn't check the implementation, there's no reason to believe the class isn't doing any protection, as even the docstring says it does. Just like with `ProtectedBinary`, once the object is initialized as protected or unprotected during instantiation, it cannot be changed (at least based on what the object reports).

Code listing 3.4 ProtectedString constructor accepting a string as an argument. Note that no matter what protection level is set by the caller, it will always end up disabled.

```
/// <summary>
 /// Construct a new protected string. The string is initialized
 /// to the value supplied in the parameters.
 /// </summary>
 /// <param name="bEnableProtection">If this parameter is <c>true</c>,
 /// the string will be protected in memory (encrypted). If it
 /// is <c>false</c>, the string will be stored as plain-text.</param>
 /// <param name="strValue">The initial string value.</param>
 public ProtectedString(bool bEnableProtection, string strValue)
 {
    Init(bEnableProtection, strValue);
 }
 private void Init(bool bEnableProtection, string str)
 {
     if(str == null) throw new ArgumentNullException("str");

     m_bIsProtected = bEnableProtection;

     // As the string already is in memory and immutable,
     // protection would be useless
     m_strPlainText = str;
 }
```

Unprotected data can be obtained with `ReadString`, `ReadChars`, and `ReadUtf8` methods. Calling `ReadString` also results in the `ProtectedString` permanently switching to unprotected mode and storing data as a regular string from that point on. This makes sense as, from that point on, control over the data has been lost.

Code listing 3.5 Author's code comments explaining why should the memory protection be disabled when the data passes through a string.

```
// As the text is now visible in process memory anyway,
// there's no need to protect it anymore (strings are
// immutable and thus cannot be overwritten)
m_strPlainText = str;
m_pbUtf8 = null;
```

`ProtectedString` also implements methods for modifying the underlying strings and concatenating them. The `Remove` method effectively allows erasing a part of the string by creating a new, shorter, `ProtectedString`. All intermediate values are deleted correctly, but the original object stays in memory because, just like `ProtectedBinary`, `ProtectedString` doesn't implement `IDisposable`, a destructor, `Clear`, or any other method for securely erasing it. And again, this means that there is no way to clear sensitive data from memory once they are not needed. Overloaded `operator +` accepts a managed string as an argument, so consistently with previously discussed methods, it also permanently disables the protection when called. Finally, while the `Insert` method also accepts a managed string, it is carefully written to not expose the data already present. This is used to build up `ProtectedString` (explored later).

A strange aspect of the implementation is that while a user of the object can use the `bool IsProtected` property to check whether the object is protected, this does not reflect reality. As was explained previously, there are occasions when the class just switches to plaintext

and gives up on protecting the data. However, `IsProtected` will always return the same initial value the `ProtectedString` was set up with, so it's impossible to know this happened.

The class is correctly using the lock statement, together with Interlocked features, to stay thread safe.

3.3.1 Usage and GUI Components

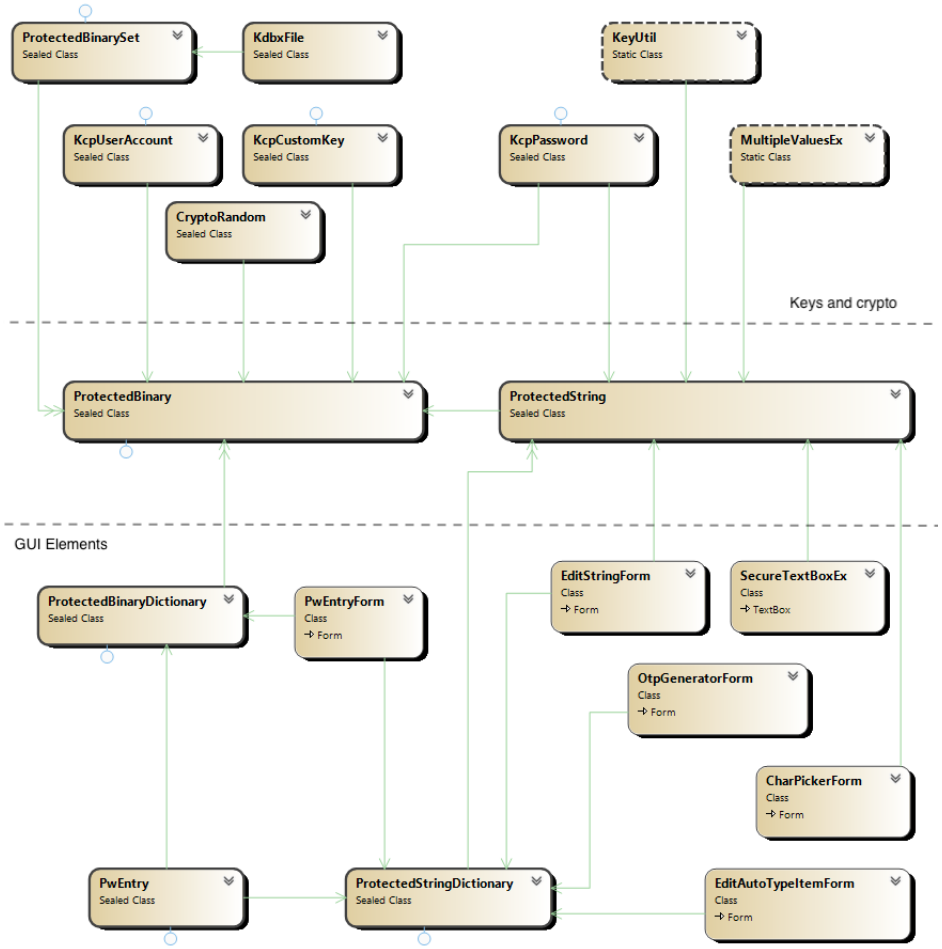
Just like with `ProtectedBinary`, when it comes to **accidental exposure**, some areas are not problematic, like the object not overloading methods like `ToString` or not being serializable. This should be effective in preventing mistakes like writing this object to logs or a file. However, it is also possible to easily create `ProtectedStrings` without any protection, possibly by mistake.

`ProtectedString` is used throughout almost the entire application 3.1. If not counting the code responsible for converting the database to/from 3rd party format (protection doesn't make much sense there), the `ProtectedString` constructor is used 49 times in 22 files. The vast majority of those use cases are related to data using `ProtectedString` in an unprotected mode, such as username, URL, notes, and other metadata. It is not clear why isn't a regular string used instead, as the level of security is identical. There are four calls of the constructor that result in a protected mode being used, two of which are from the password generator component; one protects the master password and one all the other passwords. These use cases are the main focus of the following analysis.

Another way of obtaining a `ProtectedString` instance is by calling the `WithProtection` method, with returns a new `ProtectedString` with the same data, but a new protection level. It is mainly used as a helper - shorthand for creating a new `ProtectedString` with a constructor. As the author correctly notes in the code comment, calling this method means that either the previous string was unprotected or the new one is, so the data is most likely exposed.

`SecureTextBoxEx` is a custom UI element (Windows Form) inherited from `Windows.Forms.TextBox`. Its main goal is to allow the user to provide a password securely without it staying unprotected in memory. It is likely a replacement for `Windows.Controls.PasswordBox` (returns `SecureString`). `SecureTextBoxEx` is used throughout the application for all kinds of secrets entry. As expected, it uses ProtectedString as the secret store, and characters are replaced with black circles (•). The mechanism it uses to build the `ProtectedString` is, in principle, analogous to Microsoft's `SecureString` documentation example, although less secure. Diagram 3.2 provides an overview of this process. First, the user types a character into the text box. This triggers an event handler `OnTextChanged`, which identifies the change that has been made to the text and calls a private function `RemoveInsert`, to make the change. `RemoveInsert` calls `ProtectedString`'s `Insert` to effectively add that one new character to it, and the old `ProtectedString` is replaced with this newly created one. `OnTextChanged` finishes by calling `ShowCurrentText`, which produces a string of • with the same length as the current `ProtectedString`. This is then set as the actual Text property, which automatically displays it to the user. `SecureTextBoxEx` also allows disabling the protection altogether and showing the plaintext instead. For that, the original `Windows.Forms.TextBox` property `UseSystemPasswordChar` is used. If it is set to false, the `ReadString` is called on the `ProtectedString` to obtain the plaintext and display it. This feature is commonly used with a button next to the textbox, where its purpose is to show the password to the user on demand. `SecureTextBoxEx` correctly prevents serialization of the object with the `NonSerialized` attribute, and by defining default values with `ShouldSerialize` and `Reset` methods.

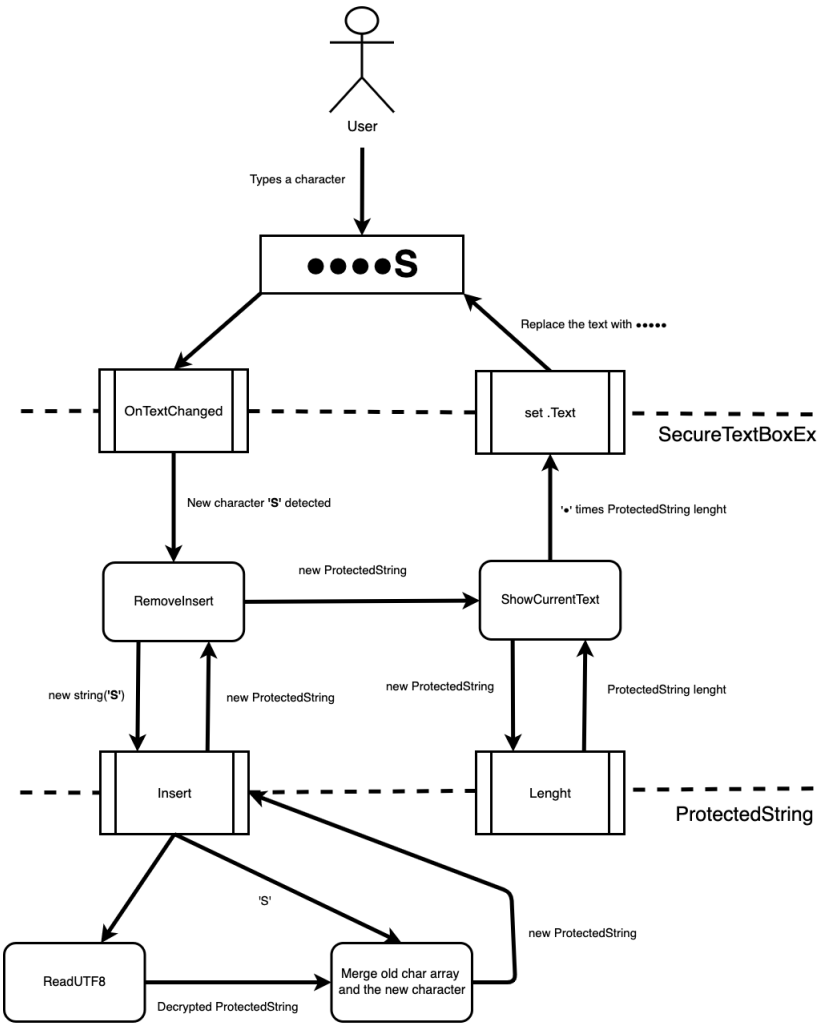
Figure 3.1 Diagram showing relationships between select KeePass classes. An from A to B indicated that class A contain an instance of class B as a property.



Analysis of this UI element was conducted on the master password input form 3.3. This form is shown right when KeePass is launched, and its purpose is to take the master password, use it to derive a key, and decrypt the database with it. This is the most important password in the entire application, so it's fitting to assess the security of this component on it. However, the following findings apply to all use cases of the element.

By conducting multiple process memory dumps and also analyzing the application in a debugger, it was discovered that `SecureTextBoxEx` might provide protection under certain conditions but fail in others. Provided the user types password characters one at a time and then presses the OK button, the entire plaintext password indeed never appears unprotected in memory and will always be stored in a `ProtectedString` until the application is shut down. This might be surprising because the `Insert` method accepts a regular string. However, since this string is always only one character long, the worst-case scenario is that for a 16-character password, there will be 16 one-character strings scattered in memory. During the tests, this proved useless to the attacker. The strings weren't near each other or ordered in memory; there was only one instance of each of them, and some of them got overwritten before the user pressed the OK button. This was confirmed on Windows 10 and 11 (ARM64). While this might change across different versions of the CLR, it provides a reasonable assurance that it likely isn't an issue.

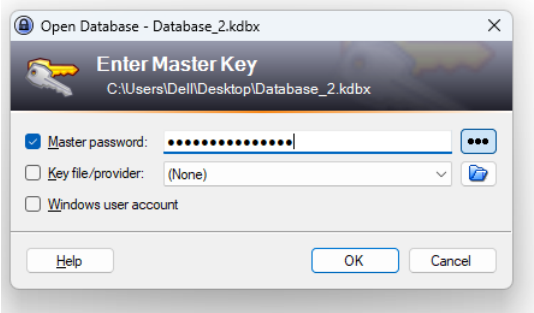
Figure 3.2 High level overview of how `SecureTextBoxEx` operates.



3.3.1.1 Vulnerability

However, `SecureTextBoxEx` is dragged down by its `Windows.Forms.TextBox` parent. As described earlier, every time the user presses a key, it is processed and then replaced by •. This means that if there was a string •••• in the text box and the user wrote the letter S there, now there's ••••S in the text box. And because this is stored as a regular string, it won't be deleted. Furthermore, the way the string is handled on its way from the text box to the handler, where it's manipulated, results in at least three copies being created! So even if one of the instances gets overwritten, there's still a high likelihood of it staying in memory. Except for the first character, it is possible to exploit this behavior and search a process dump for these patterns. Then, a list of possible candidates for each password character position can be produced. A simple application was developed to demonstrate this 3.6. No code execution on the target system is needed. There's also no need to have access to system memory to bypass DPAPI on Windows. **All that's needed is a process memory dump**, or any other file that contains that memory (hibernation file, swap file, ...). It was verified that the master password in this representation stays in memory for a long time after it was entered, **even after the database has been locked** (and the program requires the master password again). **This vulnerability has been reported. It**

Figure 3.3 KeePass master password input text box using the internal class `SecureTextBoxEx`.



got acknowledged by the author of KeePass, Dominik Reichl and hopefully will be fixed soon.

Code listing 3.6 Example output of a proof-of-concept application exploiting `SecureTextBoxEx` to dump KeePass master password from memory.

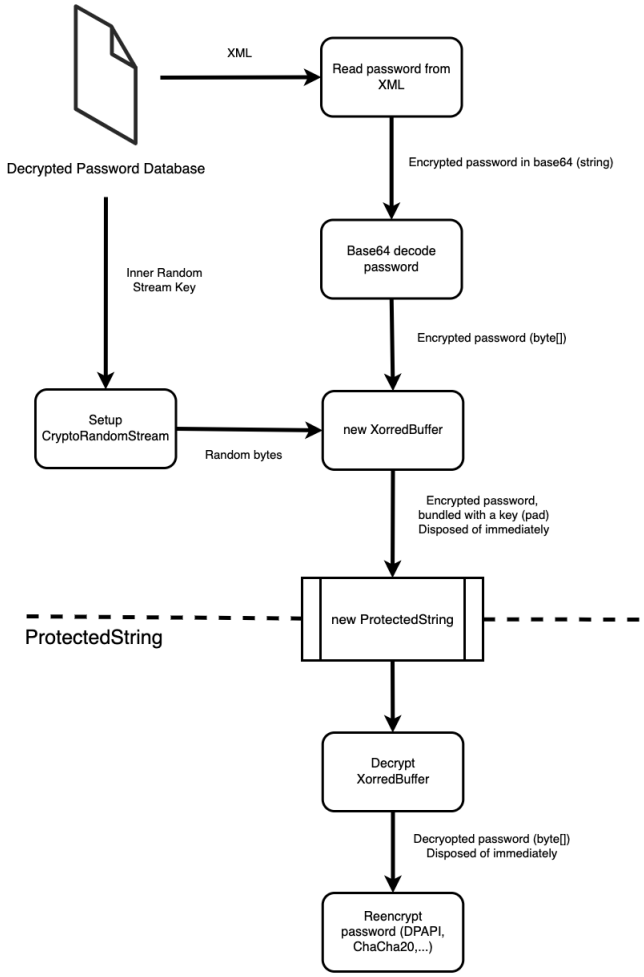
```
./keepass_password_dumper KeePass.DMP
Found: *e
Found: **c
Found: **c
...
Found: ****************o
Found: ******************r
Found: ********************d
...
Password candidates (character positions):
1.:  Unknown
2.:  e, N, c, d, f, g, W, 1,
3.:  c,
4.:  r,
5.:  e,
...
13.: P,
14.: a,
15.: s,
16.: s,
17.: w,
18.: o,
19.: r,
20.: d,
Combined: {UNKNOWN}{e, N, c, d, f, g, W, 1}cretMasterPassword
```

If the user doesn't type the password one character at a time but pastes it from the clipboard instead, it is even easier for the attacker. In that case, there will be at least three instances of that password in memory as a plaintext string just because of the `Windows.Forms.TextBox` behavior. Also, since `ProtectedString.Insert` accepts strings as well, another instance of the string will be created afterward.

Finally, `SecureTextBoxEx` also allows the revealing of the password to the user by clicking a button. Once that happens, the string again passes through multiple functions, leaving numerous plaintext instances in memory for a long time.

After the master password has been entered and the database decrypted, KeePass loads the passwords. The previously mentioned `XorredBuffer` is the key to password security here. Diagram 3.4 provides a simplified overview of the whole password-loading process. First, the Inner Random Stream Key is read from the decrypted database and is used to seed the random generator inside `CryptoRandomStream`. The key is called "inner," as opposed to outer because this is effectively a second layer of encryption. `CryptoRandomStream` is a custom class that uses various encryption algorithms to generate a cryptographically secure random bit steam. Implementation details aren't important here, apart from the fact that it implements `IDisposable` and carefully works with the key to prevent unnecessary leakage.

Figure 3.4 KeePass loading passwords into memory. Note that the string used to pass the parsed XML data doesn't matter, because at that time the data is encrypted.



The password is parsed from an XML document into a regular string. This might sound problematic, but at this stage, the password is in base64 and encrypted with a one-time pad. After it is decoded from base64, the corresponding pad is obtained from `CryptoRandomStream` for decrypting this very password (passwords are always encrypted and decrypted in the same order, so RNG with the same "seed" produces the same pad). The password still isn't decrypted but instead bundled in the `XorredBuffer` buffer, together with the pad. This `XorredBuffer` is then passed to the corresponding `ProtectedString` constructor, which decrypts it, re-encrypts it, and

stores it. `XorredBuffer` is then cleared immediately (it implements `IDisposable`), as well as the `CryptoRandomStream` and the key (after all passwords are loaded).

The passwords are now protected and loaded in memory. Numerous tests with a debugger and memory dumper were conducted to find any leaks in this process, but no issues were identified. `byte[]` is always used for unprotected passwords, so none of them stay in memory.

While the process of loading passwords is sound and well-designed, it is, again, let down by the user interface. Almost any interaction with passwords means permanent leakage of plaintext to memory. This includes obvious examples discussed previously, such as displaying the password with a dedicated button, as well as less obvious ones, such as copying it to the clipboard with a context menu.

It seems that the author is aware of some of the problems and mentions them on the KeePass website: *"For some operations, KeePass must make sensitive data available unencryptedly in the process memory. For example, in order to show a password in the standard list view control provided by Windows, KeePass must supply the cell content (the password) as unencrypted string (unless hiding using • is enabled). Operations that result in unencrypted data in the process memory include, but are not limited to: displaying data (not •) in standard controls, searching data, replacing placeholders (during auto-type, dragdrop, copying to clipboard, ...), importing/-exporting files (except KDBX) and loading/saving unencrypted files. Windows and .NET may make copies of the data (in the process memory) that cannot be erased by KeePass."*

3.4 ProtectedData, ProtectedMemory, and Other Protection Measures

As discussed previously, ProtectedBinary relies on ProtectedMemory on Windows. Apart from that, KeePass doesn't seem to be using any other protection measures introduced in this thesis. ProtectedData is present in the code, but it doesn't seem to be actively used.

3.5 Known Attacks on KeePass

For KeePass, like for many popular password managers, offensive tooling exists. KeeThief [72] works by attaching to the running KeePass process with `Microsoft.Diagnostics.Runtime`. Once it finds the encrypted master key, it decrypts it by injecting shellcode into the KeePass process. This shellcode calls `RtlDecryptMemory`, and since it's the same process, the memory is successfully decrypted. KeeFarce [73] uses a different injection method, but the approach is similar. It injects a DLL into the KeePass process and then calls an internal KeePass method that exports the entire database (no need to deal with DPAPI separately).

These and similar tools need to be able to execute code on the target system to work.

3.6 Conclusion

It is obvious that KeePass could use some improvement when it comes to memory protection. While designing a custom set of objects was beneficial in some cases, it was detrimental in others. The main benefit is that unlike `SecureString`, `ProtectedString`, and `ProtectedBinary` provide at least some level of protection on Linux and macOS. On the other hand, secrets are being stored in managed byte arrays, and there's no way to dispose of either of these objects once they are created. The confusing class design could lead to problems in the future since it's easy to create unprotected objects by mistake. It would probably have been beneficial to start with `SecureString` source code instead and add macOS/Linux support to it, than attempt to design something from scratch. The same goes for using custom implementations of encryption

algorithms and cryptographic primitives. However, it is important to note that most of the issues identified on this level don't have a tangible impact, or it's hard to create a meaningful proof-of-concept around them (e.g., managed byte arrays). Furthermore, the mechanism of loading passwords from the database into memory is well-designed and, at least on Windows, provides a reasonable level of protection.

The situation becomes more complicated when it comes to UI. Despite the author making the best effort and developing custom UI elements, it is trivial to obtain the master password from memory, even without executing any code on the machine. While it certainly would be possible to develop more secure alternatives to Windows Forms UI elements, it might be hard to justify such an effort. If regular software developers are required to write everything from scratch to do things right, what is the point of frameworks? High-level languages, in general, aren't well suited for working with secrets, and without great support from both the language and the operating system, it might be nearly impossible to do everything correctly.

That's why it is difficult to blame the author of KeePass for these shortcomings. Furthermore, based on the comments both in the code and on the website, the author is well aware of most of these issues and the reported vulnerability will hopefully get fixed soon.

Of course, in theory, and with unlimited resources, there are many areas that would help improve memory protection in KeePass. On macOS and Linux, KeePass could utilize native services to store memory encryption keys (Secure Enclave, Keychain, Gnome Keyring, ...). UI elements could be designed from scratch to never leak data, and KeePass could utilize dedicated hardware, like a TPM. All of this means going beyond what .NET Framework provides.

Chapter 4
Conclusion

The .NET ecosystem is vast and convoluted, and this is also reflected in its memory protection capabilities. Depending on the platform and specific use case, it might be reasonably effective or completely insufficient. In order for the protection to be effective, developers need to understand how memory management works and what tools are available to change its behavior. The experiments described in this thesis demonstrated that even small mistakes can be costly.

The best protection is offered on Windows with the .NET Framework. Still, the protection is exclusively based on software. Today's hardware offers promising capabilities that could be used for memory protection. However, at least in the case of .NET, this is irrelevant because none of it is made available to developers.

Analysis of the protection mechanisms discovered that, on their own, they are well-designed and provide a reasonable level of protection (on Windows). This is especially true for SecureString. However, problems arise when it comes to interoperability. SecureString isn't supported in most of the .NET classes, let alone the operating system or 3rd party libraries. Apart from the PasswordBox in WPF, .NET UI frameworks lack support for SecureString, potentially rendering all underlying protections useless. This means that additional code often needs to be developed, increasing the barrier for developers. On platforms outside of Windows, protection is almost non-existent, and developers have to look for alternatives outside .NET.

Fundamentally, high-level languages aren't very suitable for protecting sensitive data in memory. In order for the protection to be truly effective, developers are required to tightly manage resources, similarly to how it is done in low-level languages like C.

KeePass provides a good example of how these protections end up being used in the real world. Despite the author's best efforts, it has proven difficult to have a cross-platform solution that works well on all of them. It might be unreasonable to expect regular developers to go to such lengths just to guarantee basic security for their applications.

Apart from becoming easier to use, it is evident that in order for memory protection to become effective, tight integration of the hardware, operating system, and software is needed. The current shift to ARM CPUs, which commonly have Trusted Execution Environments, will hopefully be a turning point for this situation.

Bibliography

1. ACUNA, Armando. Genius Unplugs, Puts Life on Hold : Famed Computer Whiz, Hacker Spurns Money for Existence on Street. *Los Angeles Times*. 1989. Available also from: https://www.latimes.com/archives/la-xpm-1989-03-19-me-297-story.html.
2. *OWASP Top Ten 2017 | A3:2017-Sensitive Data Exposure | OWASP Foundation*. 2022. Available also from: https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure. [Online; accessed 1. Apr. 2023].
3. DIVERTO. *Extracting passwords from hiberfil.sys and memory dumps*. 2019. Available also from: https://diverto.github.io/2019/11/05/Extracting-Passwords-from-hiberfil-and-memdumps. [Online; accessed 2. Mar. 2023].
4. *CVE-2014-0160*. [Available from MITRE, CVE-ID CVE-2014-0160.]. 2013. Available also from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. [Online; accessed 2. Mar. 2023].
5. CARMAA. *Inception - physical memory manipulation and hacking tool exploiting PCI-based DMA*. 2023. Available also from: https://github.com/carmaa/inception. [Online; accessed 2. Mar. 2023].
6. *A Detailed Analysis of the RedLine Stealer*. 2023. Available also from: https://securityscorecard.com/research/detailed-analysis-redline-stealer. [Online; accessed 2. Mar. 2023].
7. COSTAN, Victor; DEVADAS, Srinivas. *Intel SGX Explained* [Cryptology ePrint Archive, Paper 2016/086]. 2016. Available also from: https://eprint.iacr.org/2016/086. https://eprint.iacr.org/2016/086.
8. *SGX 101, The very first place to study Intel SGX*. 2023. Available also from: https://sgx101.gitbook.io/sgx101/sgx-bootstrap/overview. [Online; accessed 4. Apr. 2023].
9. *Using Intel's SGX to keep secrets even safer | 1Password*. 2017. Available also from: https://blog.1password.com/using-intels-sgx-to-keep-secrets-even-safer. [Online; accessed 2. Apr. 2023].
10. *Intel® Software Guard Extensions (Intel® SGX) Web-Based Training*. 2023. Available also from: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sgx-web-based-training.html. [Online; accessed 5. Mar. 2023].
11. NILSSON, Alexander; BIDEH, Pegah Nikbakht; BRORSSON, Joakim. *A Survey of Published Attacks on Intel SGX*. 2020. Available from arXiv: 2006.13598 [cs.CR].
12. SCHWARZ, Michael; WEISER, Samuel; GRUSS, Daniel; MAURICE, Clémentine; MANGARD, Stefan. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *CoRR*. 2017, vol. abs/1702.08719. Available from arXiv: 1702.08719.

13. *Enclave Signing Tool for Intel® Software Guard Extensions (Intel® SGX)*. 2023. Available also from: https://www.intel.com/content/www/us/en/content-details/671205/enclave-signing-tool-for-intel-software-guard-extensions-intel-sgx.html. [Online; accessed 6. Mar. 2023].

14. *AMD Secure Encrypted Virtualization (SEV)*. 2023. Available also from: https://www.amd.com/en/developer/sev.html. [Online; accessed 6. Mar. 2023].

15. MÜLLER, Tilo; FREILING, Felix; DEWALD, Andreas. TRESOR runs encryption securely outside RAM. In: 2011, pp. 17–17.

16. TRUSTED COMPUTING GROUP. *TPM 2.0 Library Specification*. 2019-11. Available also from: https://trustedcomputinggroup.org/resource/tpm-library-specification/. [Online; accessed 5. Apr. 2023].

17. *CVE-2023-1017*. [Available from MITRE, CVE-ID CVE-2023-1017.]. 2023. Available also from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-1017. [Online; accessed 6. Mar. 2023].

18. *CVE-2023-1018*. [Available from MITRE, CVE-ID CVE-2023-1018.]. 2023. Available also from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-1018. [Online; accessed 6. Mar. 2023].

19. *Sniff, there leaks my BitLocker key*. 2021. Available also from: https://labs.withsecure.com/publications/sniff-there-leaks-my-bitlocker-key. [Online; accessed 7. Mar. 2023].

20. *Unlocking BitLocker: Can You Break That Password?* 2020. Available also from: https://blog.elcomsoft.com/2020/05/unlocking-bitlocker-can-you-break-that-password. [Online; accessed 3. Apr. 2023].

21. *Breaking VeraCrypt: Obtaining and Extracting On-The-Fly Encryption Keys*. 2021. Available also from: https://blog.elcomsoft.com/2021/06/breaking-veracrypt-obtaining-and-extracting-on-the-fly-encryption-keys. [Online; accessed 3. Apr. 2023].

22. MICROSOFT. *How Windows uses the TPM*. 2023. Available also from: https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm. [Online; accessed 4. Apr. 2023].

23. MICROSOFT. *CngProvider Class (System.Security.Cryptography)*. [N.d.]. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.cngprovider?view=net-7.0. [Online; accessed 6. Apr. 2023].

24. MICROSOFT. *TSS.MSR - TPM Software Stack (TSS) implementations from Microsoft*. 2022. Available also from: https://github.com/Microsoft/TSS.MSR. [Online; accessed 2. Apr. 2023].

25. *About Windows Hello security in 1Password for Windows*. 2022. Available also from: https://support.1password.com/windows-hello-security. [Online; accessed 7. Apr. 2023].

26. MICROSOFT. *Windows Data Protection*. 2001. Available also from: https://learn.microsoft.com/en-us/previous-versions/ms995355(v=msdn.10). [Online; accessed 6. Apr. 2023].

27. PICOD, Jean-Michel; BURSZTEIN, Elie. Reversing dpapi and stealing windows secrets offline. In: 2010.

28. MICROSOFT. *Dpapi.h header - Win32 apps*. 2023. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/dpapi. [Online; accessed 10. Apr. 2023].

29. MICROSOFT. *CNG DPAPI - Win32 apps*. 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/seccng/cng-dpapi. [Online; accessed 10. Apr. 2023].

30. *Windows 11 TPM Protection, Passwordless Sign-In and What You Can Do About Them*. 2022. Available also from: https://blog.elcomsoft.com/2022/03/windows-11-tpm-protection-passwordless-sign-in-and-what-you-can-do-about-them. [Online; accessed 10. Apr. 2023].

31. GENTILKIWI. *mimikatz*. 2023. Available also from: https://github.com/gentilkiwi/mimikatz. [Online; accessed 10. Apr. 2023].

32. MICROSOFT. *CryptProtectMemory function (dpapi.h) - Win32 apps*. 2022. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/dpapi/nf-dpapi-cryptprotectmemory. [Online; accessed 10. Apr. 2023].

33. MICROSOFT. *CryptUnprotectMemory function (dpapi.h) - Win32 apps*. 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/dpapi/nf-dpapi-cryptunprotectmemory. [Online; accessed 10. Apr. 2023].

34. MICROSOFT. *RtlEncryptMemory function (ntsecapi.h)*. 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlencryptmemory. [Online; accessed 10. Apr. 2023].

35. GTWOREK. *$IOCTL_UNPROTECT$.c*. 2023. Available also from: https://github.com/gtworek/PSBits/blob/master/Misc/IOCTL_UNPROTECT.c. [Online; accessed 10. Apr. 2023].

36. TPM2-SOFTWARE. *tpm2-tss*. 2023. Available also from: https://github.com/tpm2-software/tpm2-tss. [Online; accessed 10. Apr. 2023].

37. *Projects/GnomeKeyring - GNOME Wiki!* 2023. Available also from: https://wiki.gnome.org/Projects/GnomeKeyring. [Online; accessed 10. Apr. 2023].

38. *KWallet - KWallet Framework*. 2023. Available also from: https://api.kde.org/frameworks/kwallet/html/index.html. [Online; accessed 10. Apr. 2023].

39. *Projects/GnomeKeyring/SecurityPhilosophy - GNOME Wiki!* 2023. Available also from: https://wiki.gnome.org/Projects/GnomeKeyring/SecurityPhilosophy. [Online; accessed 10. Apr. 2023].

40. *Projects/GnomeKeyring/Memory - GNOME Wiki!* 2023. Available also from: https://wiki.gnome.org/Projects/GnomeKeyring/Memory. [Online; accessed 10. Apr. 2023].

41. MICROSOFT. *Microsoft Edge password manager security*. 2023. Available also from: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-security-password-manager-security. [Online; accessed 18. Apr. 2023].

42. *Apple Platform Security*. 2023. Available also from: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf. [Online; accessed 3. Apr. 2023].

43. *Encrypting for Apple's Secure Enclave | Darth Null*. 2022. Available also from: https://darthnull.org/secure-enclave-ecies. [Online; accessed 7. Mar. 2023].

44. *Protecting keys with the Secure Enclave | Apple Developer Documentation*. 2023. Available also from: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave. [Online; accessed 3. Apr. 2023].

45. MAXGOEDJEN. *Secretive - Secretive is an app for storing and managing SSH keys in the Secure Enclave*. 2023. Available also from: https://github.com/maxgoedjen/secretive. [Online; accessed 3. Apr. 2023].

46. *About the security of using Touch ID or Apple Watch to unlock 1Password for Mac*. 2023. Available also from: https://support.1password.com/touch-id-apple-watch-security-mac. [Online; accessed 3. Apr. 2023].

47. *CVE-2021-30784.* [Available from MITRE, CVE-ID CVE-2021-30784.]. 2021. Available also from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30784. [Online; accessed 6. Mar. 2023].
48. *About the security content of macOS Big Sur 11.5.* 2023. Available also from: https://support.apple.com/en-us/HT212602. [Online; accessed 5. Apr. 2023].
49. MCGILLION, Brian; DETTENBORN, Tanel; NYMAN, Thomas; ASOKAN, N. Open-TEE - An Open Virtual Trusted Execution Environment. *CoRR*. 2015, vol. abs/1506.07367. Available from arXiv: 1506.07367.
50. *What is .NET Framework? A software development framework.* 2023. Available also from: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework. [Online; accessed 10. Apr. 2023].
51. LOCK, Andrew. Understanding the .NET ecosystem: The evolution of .NET into .NET 7. *Andrew Lock | .NET Escapades*. 2023. Available also from: https://andrewlock.net/understanding-the-dotnet-ecosystem-the-evolution-of-dotnet-into-dotnet-7.
52. MICROSOFT. *.NET garbage collection.* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection. [Online; accessed 10. Apr. 2023].
53. MICROSOFT. *BSTR.* 2023. Available also from: https://learn.microsoft.com/en-us/previous-versions/windows/desktop/automat/bstr. [Online; accessed 10. Apr. 2023].
54. MICROSOFT. *CryptographicOperations.ZeroMemory(Span) Method (System.Security.Cryptography).* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.cryptographicoperations.zeromemory?view=net-8.0. [Online; accessed 10. Apr. 2023].
55. MICROSOFT. *ProtectedData Class (System.Security.Cryptography).* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.protecteddata?source=recommendations&view=windowsdesktop-7.0. [Online; accessed 10. Apr. 2023].
56. MICROSOFT. *CRYPT_INTEGER_BLOB structure (Windows).* 2023. Available also from: https://learn.microsoft.com/en-us/previous-versions/windows/desktop/legacy/aa381414(v=vs.85). [Online; accessed 10. Apr. 2023].
57. MICROSOFT. *ProtectedMemory Class (System.Security.Cryptography).* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.protectedmemory?view=netframework-4.8.1. [Online; accessed 10. Apr. 2023].
58. MICROSOFT. *SecureString Class (System.Security).* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.security.securestring?view=net-7.0. [Online; accessed 10. Apr. 2023].
59. MICROSOFT. *SysAllocStringLen function (oleauto.h) - Win32 apps.* 2023. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/oleauto/nf-oleauto-sysallocstringlen. [Online; accessed 10. Apr. 2023].
60. MICROSOFT. *Span Struct (System).* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.span-1?view=net-8.0. [Online; accessed 10. Apr. 2023].
61. MICROSOFT. *Marshal.SecureStringToGlobalAllocUnicode(SecureString).* 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.securestringtoglobalallocunicode?view=net-7.0. [Online; accessed 10. Apr. 2023].

62. MICROSOFT. *ConvertTo-SecureString (Microsoft.PowerShell.Security) - PowerShell*. 2023. Available also from: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/convertto-securestring?view=powershell-7.3. [Online; accessed 10. Apr. 2023].

63. MICROSOFT. *Create a Password Text Box with TextBox Control - Windows Forms .NET Framework*. 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/controls/how-to-create-a-password-text-box-with-the-windows-forms-textbox-control?view=netframeworkdesktop-4.8. [Online; accessed 10. Apr. 2023].

64. *Reference Source*. 2015. Available also from: https://referencesource.microsoft.com/#System/net/System/Net/UnsafeNativeMethods.cs,77e10d03a6d14232. [Online; accessed 10. Apr. 2023].

65. *Rfc2898DeriveBytes + PBKDF2 + SecureString is it possible to use a secure string instead of a string?* 2023. Available also from: https://stackoverflow.com/questions/9734043/rfc2898derivebytes-pbkdf2-securestring-is-it-possible-to-use-a-secure-string/43858011#43858011. [Online; accessed 10. Apr. 2023].

66. MICROSOFT. *PasswordBox Class (Windows.UI.Xaml.Controls) - Windows UWP applications*. 2023. Available also from: https://learn.microsoft.com/en-us/uwp/api/windows.ui.xaml.controls.passwordbox?view=winrt-22621. [Online; accessed 10. Apr. 2023].

67. REICHL, Dominik. *KeePass Password Safe*. 2023. Available also from: https://keepass.info. [Online; accessed 10. Apr. 2023].

68. REICHL, Dominik. *Awards/Ratings - KeePass*. 2023. Available also from: https://keepass.info/ratings.html. [Online; accessed 10. Apr. 2023].

69. REICHL, Dominik. *Security - KeePass*. 2023. Available also from: https://keepass.info/help/base/security.html. [Online; accessed 10. Apr. 2023].

70. MICROSOFT. *RNGCryptoServiceProvider Class (System.Security.Cryptography)*. 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?view=net-8.0. [Online; accessed 10. Apr. 2023].

71. MICROSOFT. *Guid.NewGuid Method (System)*. 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.guid.newguid?view=net-7.0. [Online; accessed 10. Apr. 2023].

72. GHOSTPACK. *KeeThief*. 2020. Available also from: https://github.com/GhostPack/KeeThief. [Online; accessed 10. Apr. 2023].

73. DENANDZ. *KeeFarce*. 2015. Available also from: https://github.com/denandz/KeeFarce. [Online; accessed 10. Apr. 2023].

Contents of Enclosed Media

```
README.md ..... brief description of the contents
src/ ..... sources for the proof-of-concept applications
text/ ..... thesis
    thesis.pdf ..... this thesis in PDF
```