UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA V KOŠICIACH
PRÍRODOVEDECKÁ FAKULTA

PRÍSTUPY K OŠETRENIU ÚNIKU DÁT Z PAMÄTE

2019 **Katarína AMRICHOVÁ**

UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA V KOŠICIACH
PRÍRODOVEDECKÁ FAKULTA

PRÍSTUPY K OŠETRENIU ÚNIKU DÁT Z PAMÄTE

BAKALÁRSKA PRÁCA

Študijný program:	Informatika
Pracovisko:	Ústav informatiky
Vedúci práce:	Mgr. Terézia Mézešová

Košice 2019

Katarína AMRICHOVÁ

Univerzita P. J. Šafárika v Košiciach
Prírodovedecká fakulta

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Katarína Amrichová
Študijný program:	Informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	Bakalárska práca
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický

Názov: Prístupy k ošetreniu úniku dát z pamäte

Názov EN: Approaches for mitigation of Heap Inspection vulnerability

Cieľ:
1. Analyzovať a spracovať spôsoby získania dát z pamäte alokovanej aplikáciou
2. Analyzovať a spracovať aktuálne prístupy k ošetreniu úniku dát z pamäte alokovanej aplikáciou
3. Navrhnúť, implementovať, a vyhodnotiť preventívne opatrenie voči úniku dát z pamäte na úrovni zdrojového kódu aplikácie

Literatúra:
1. Zeng, Q., Zhao, M., Liu, P.: HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows. Proc. Int. Conf. Dependable Syst. Networks. 2015–September, 485–496 (2015).
2. Focardi, R., Palmarini, F., Squarcina, M., Steel, G., Tempesta, M.: Mind Your Keys? A Security Evaluation of Java Keystores. Proc. 2018 Netw. Distrib. Syst. Secur. Symp. (2018).
3. Asad, A., Ali, H.: Working with Cryptography. In: The C Programmer's Study Guide (MCSD). pp. 347–364. Apress, Berkeley, CA (2017).
4. Mun, H.-J., Li, Y.: Secure Short URL Generation Method that Recognizes Risk of Target URL. Wirel. Pers. Commun. 93, 269–283 (2017).

Vedúci:	Mgr. Terézia Mézešová
Ústav :	ÚINF - Ústav informatiky
Riaditeľ ústavu:	prof. RNDr. Viliam Geffert, DrSc.

Dátum schválenia: 03.05.2019

Poďakovanie

Ďakujem vedúcej práce Mgr. Terézii Mézešovej za pripomienky, odbornú pomoc pri písaní práce a za venovaný čas.

Abstrakt v štátnom jazyku

Počítačové programy často pracujú s rôznymi citlivými údajmi, ktoré sa zväčša ukladajú a spracovávajú ako reťazce, pričom v objektovo orientovaných programovacích jazykoch sa na tento účel používa trieda `String`. Ukladanie citlivých údajov do objektu typu `String` však nie je bezpečné, pretože `String` je nešifrovaný a navyše sa v operačnej pamäti môže nachádzať dlho potom ako prestal byť potrebný, pričom programátor si nedokáže vynútiť jeho vymazanie – za to je zodpovedný `Garbage Collector`, ktorý sa správa veľmi nedeterministicky. Ak sa `String` obsahujúci citlivé údaje nachádza v operačnej pamäti a útočník sa dostane buď k jej časti alebo priamo k celému obrazu, dokáže tieto citlivé údaje poľahky prečítať. Táto práca sa zaoberá útokmi s týmto cieľom a analyzujú sa v nej existujúce riešenia voči podobným útokom. Výsledkom našej práce je pseudokód, ktorý popisuje štruktúru a fungovanie triedy `SecureString` slúžiacej ako bezpečná alternatíva ku `String`-u. `SecureString` šifruje citlivé údaje, ktoré uchováva, taktiež poskytuje metódy na ich úpravu a porovnávanie a dovoľuje programátorovi úplne odstrániť citlivý obsah z operačnej pamäte, kedykoľvek prestane byť potrebný. Náš návrh je kompatibilný so štandardom Payment Card Industry Data Security Standard (PCI DSS), je teda možné používať ho aj na spracúvanie údajov súvisiacich s platobnými kartami.

Kľúčové slová: Citlivé údaje, Ochrana údajov v pamäti, Únik údajov, Secure String

Abstrakt v cudzom jazyku

Computer programs often work with a variety of sensitive data, which are mostly stored and processed as a series of characters and class `String` is widely used in object-oriented programming languages for this purpose. However, saving sensitive data to a `String` object is not safe as `String` is not encrypted and may still be in the operating memory even after it is no longer needed. Moreover, the programmer cannot enforce its erasure – a mechanism called `Garbage Collector` is responsible for removing unused items from the operating memory. However, `Garbage Collector` works very unpredictably, therefore we cannot say with certainty when String with sensitive data will actually be removed from memory. If `String` containing sensitive data exists in operating memory and the attacker gets either part of it or even the entire memory image, then he can easily read these sensitive data. This thesis deals with attacks having such objective and analyzes existing countermeasures against them. The result of our work is pseudocode that describes the structure and functionality of the `SecureString` class which can be used as a safe alternative to a classic `String`. `SecureString` encrypts the sensitive data it holds, it also provides methods for editing and comparing them and allows the programmer to completely remove sensitive content from operating memory whenever storing of such data becomes unnecessary. Our solution is compatible with the Payment Card Industry Data Security Standard (PCI DSS), therefore it can be used to process payment card related data.

Keywords: Sensitive data, Data protection in memory, Data leaks, Secure String

Obsah

Úvod

Počítačové programy pracujú s mnohými citlivými údajmi, ako sú napríklad heslá, šifrovacie kľúče, čísla kreditných kariet a podobne, pričom je úlohou vývojára, aby program ochránil tieto citlivé údaje a zabezpečil ich integritu, dôvernosť a dostupnosť. Tieto citlivé údaje sú zväčša v textovej forme a sú spracované ako reťazce. Na ukladanie a prácu s reťazcami sa v objektovo orientovaných programovacích jazykoch (Java, C#, C++,…) používajú objekty typu `String`. Hlavným problémom pri ukladaní citlivých údajov do objektov typu `String` je, že za vymazanie týchto objektov zodpovedá `Garbage Collector`, ktorý sa správa veľmi nedeterministicky. Programátor si nedokáže vynútiť vymazanie objektu typu `String` a preto ak skúsený útočník preskúma obsah operačnej pamäte, môže poľahky odhaliť `String` s citlivými údajmi, ktorý je v nej uložený.

Existuje nespočetne veľa útokov, pomocou ktorých útočník môže preskúmať obsah operačnej pamäte. Útoky majúce za cieľ získanie citlivých údajov z pamäte môžeme rozdeliť na *hardvérové*, na prevedenie ktorých musí mať útočník fyzický prístup k počítaču a na *softvérové*, ktoré môže útočník vykonať aj na diaľku, napríklad podsunutím vhodne upravených vstupov. Pomocou hardvérových útokov sa väčšinou získavajú dáta vyhotovením obrazu operačnej pamäte, prípadne odpočúvaním komunikácie komponentov počítača po určitých zberniciach. Softvérové útoky väčšinou zneužívajú neošetrené pretečenia premenných, zraniteľnosti v návrhu softvéru ale aj zraniteľnosti v celkovom návrhu architektúry moderných počítačov. Pomocou softvérových útokov môže útočník získať vyššie privilégiá a vykonávať škodlivý kód alebo pomocou nich môžu uniknúť časti pamäte obsahujúe dáta citlivého charakteru. V tejto práci analyzujeme aktuálne útoky na získavanie citlivých údajov z pamäte, ale rovnako analyzujeme aj známe a preskúmané útoky, ktoré stále predstavujú reálnu hrozbu.

Na zamedzenie podobným útokom existujú rôzne prístupy. Mnohé riešenia sú na úrovni operačného systému a majú za cieľ skomplikovať útočníkovi získavanie citlivých údajov napríklad pomocou automatickej detekcie a prevencie pretečení premenných alebo pomocou hardvérového šifrovania citlivých údajov, či dokonca šifrovania celej operačnej pamäte. Rovnako sú známe aj riešenia na aplikačnej úrovni, ktoré sa snažia ochrániť citlivé údaje v rámci danej aplikácie. Jedná sa prevažne o rôzne

triedy v objektovo orientovaných programovacích jazykoch, ktoré umožňujú bezpečné spracovanie v nich uložených citlivých údajov. Tieto triedy však v niektorých programovacích jazykoch nie sú dostatočne prepracované a buď neposkytujú všetku potrebnú funkcionalitu alebo ich spracovanie a ochrana citlivých údajov nie je na dostatočnej úrovni. Z toho dôvodu sme sa rozhodli existujúce triedy analyzovať a na základe analýzy vybrať najvhodnejšie a najbezpečnejšie prístupy k spracovaniu v nich uložených citlivých údajov, aby sme potom za pomoci vybratých vlastností a atribútov v pseudokóde navrhli bezpečnú alternatívu k objektu `String`. Táto bezpečná alternatíva slúži ako predloha, na základe ktorej môže byť v ľubovoľnom programovacom jazyku implementovaná trieda bezpečne spracúvajúca citlivé údaje. Na základe nášho pseudokódu taktiež môžu byť triedy, ktoré už v jednotlivých programovacích jazykoch existujú, vylepšené, bezpečnejšie a použiteľnejšie.

Po vytvorení pseudokódu triedy `SecureString` kontrolujeme jeho kompatibilitu s požiadavkami Payment Card Industry Data Security Standard (PCI DSS) štandardu. Tento štandard sú povinné dodržiavať a spĺňať všetky entity spracúvajúce, ukladajúce alebo preposielajúce údaje viažúce sa k platobnej karte, k jej držiteľovi alebo autentifikačné údaje. Na to, aby boli programátori oprávnení použiť implementáciu nášho návrhu pri vytváraní aplikácie spracúvajúcej citlivé údaje súvisiace s platobnými kartami, musí byť tento návrh kompatibilný práve s PCI DSS štandardom.

V prvej kapitole popisujeme, ako objektovo orientované programovacie jazyky pracujú s objektom typu `String` (ako ho ukladajú do operačnej pamäte, kedy ho z nej vymažú, atď.). Pritom sa sústredíme na to, aký dopad má takéto zaobchádzanie s objektom typu `String` na bezpečnosť v ňom uložených citlivých údajov. V druhej kapitole našej práce skúmame rôzne útoky na získavanie citlivého obsahu z operačnej pamäte. V tretej kapitole skúmame existujúce riešenia na ochranu citlivých údajov pred týmito útokmi. V štvrtej kapitole analyzujeme existujúce riešenia na aplikačnej úrovni a to konkrétne implementácie bezpečnej alternatívy k objektu typu `String` v troch rôznych často používaných objektovo orientovaných programovacích jazykoch. V piatej kapitole vysvetľujeme ako sme na základe týchto analyzovaných implementácií vytvorili pseudokód, ktorý popisuje ako by mala vyzerať bezpečná alternatíva k objektu typu `String`. V šiestej kapitole analyzujeme, či pseudokód `SecureString` spĺňa štandard PCI DSS.

1 String v jazykoch s manažovanou pamäťou

Podľa [1] hlavným problémom objektu `String` je, že je nezmeniteľný (immutable) a nie je pripnutý v pamäti (pinned). Keďže je nezmeniteľný, tak pokiaľ mu priradíme prázdny reťazec v snahe vynulovať citlivé údaje, ktoré predtým uchovával, v skutočnosti jeho obsah nevymažeme. Vytvoríme iba nový prázdny `String`, na ktorý odteraz bude pôvodná premenná ukazovať. V operačnej pamäti sa pôvodný citlivý obsah bude stále nachádzať a to až pokým ho odtiaľ neodstráni `Garbage Collector`. `Garbage Collector` rozhodne, kedy bude objekt vymazaný po tom, ako pre danú aplikáciu prestal byť potrebný.

Zároveň reťazec nie je pripnutý v pamäti, teda môže byť v rámci manažovania pamäte presúvaný a bez vedomia programátora môžu podľa [1] v rámci optimalizácií vznikať rôzne jeho kópie, nad ktorými programátor nemá kontrolu. Čím viac kópií citlivých údajov sa nachádza v pamäti, tým väčšia je pravdepodobnosť, že aspoň jedna z nich bude odhalená útočníkom. Ten nemusí mať prístup k celej pamäti, ale iba k nejakej jej časti, lenže čím viac kópii existuje, tým pravdepodobnejšie bude jedna z nich zapísaná práve v uniknutej/kompromitovanej časti pamäte.

Ďalším problémom je, že citlivé údaje ostávajú v pamäti dlho potom, ako boli použité aplikáciou. Stáva sa to najmä v jazykoch s manažovanou pamäťou (memory managed), v ktorých funguje mechanizmus nazývaný `Garbage Collector`. Programátor vymazanie objektu z pamäte nevie vo veľkej miere ovplyvniť, rozhoduje o ňom výlučne `Garbage Collector`. Niekedy trvá dlhú dobu, kým je nepotrebný `String` s citlivými údajmi odstránený. Čím dlhšie je uchovávaný v pamäti, tým väčšiu šancu má útočník, aby sa dostal k jeho obsahu. Podľa [2] aby `Garbage Collector` začal zbierať, musí byť splnená jedna z nasledujúcich podmienok:

- systém má málo fyzickej pamäte
- pamäť pridelená objektom na halde prekročí povolený prah (prah je automaticky nastavovaný algoritmom `Garbage Collector`-a)
- je zavolaná metóda `GC.Collect` (nepoužíva sa často – hlavne v špecifických prípadoch, napríklad pri testovaní)

Ak aj `Garbage Collector` začne zbierať, nevieme určiť, kedy a či vôbec odstráni náš už nepotrebný `String` s citlivými údajmi. V prvom rade, `Garbage Collector` smie zbierať iba také objekty, na ktoré už neexistuje referencia. Podľa [3] existujú

nasledujúce spôsoby, ako urobiť objekt vhodný na zozbieranie `Garbage Collector`-om.

- ak bol vytvorený v rámci metódy, tak je objekt vhodný pre `Garbage Collector` hneď, ako sa metóda vykoná
- referencii na objekt môže byť priradená hodnota `NULL` alebo iný objekt, ktorý bude odteraz referovať a týmpádom ak na pôvodný objekt neexistuje už žiadna iná referencia, môže byť zozbieraný `Garbage Collector`-om
- ak bol objekt vytvorený ako anonymný (nebola mu priradená nijaká referencia), tiež môže byť zozbieraný `Garbage Collector`-om

Skutočnosť, či daný objekt bude zozbieraný `Garbage Collector`-om závisí aj od toho, ktorá generácia je mu pridelená. Rozdelenie do generácií je popísané v [2] takto:

- generácia 0: je to najmladšia generácia, ktorá obsahuje objekty s krátkou životnosťou ako sú napríklad dočasné premenné
- generácia 1: tiež obsahuje objekty s krátkou životnosťou a slúži ako medzistupeň medzi generáciou 1 a generáciou 2
- generácia 2: obsahuje objekty s dlhou životnosťou ako napríklad objekty so statickými dátami, ktoré sú využívané počas celého behu procesu

Objekty, ktoré boli čerstvo alokované majú implicitne nastavenú generáciu 0, pokiaľ to nie sú veľké objekty (majúce veľkosť aspoň 85 000 bajtov), ktoré majú automaticky nastavenú generáciu 2. Keď objekt prežije čistenie pamäte `Garbage Collector`-om, tak sa automaticky posúva do vyššej generácie. Ak je detegované, že prežilo príliš veľa objektov v generácii, bude zvyšený prah pre pamäť pridelenú objektom na halde, aby sa ďalší beh `Garbage Collector`-a spustil až neskôr.

2 Útoky na získanie citlivých údajov z pamäte RAM

V tejto kapitole skúmame spôsoby získavania dát z pamäte alokovanej aplikáciou a útoky s tým spojené. Je potrebné spoznať tieto útoky, ich princípy a mechanizmy, aby sme na základe takej analýzy vedeli proti nim navrhnúť účinnú ochranu. Na uskutočnenie hardvérového útoku útočník potrebuje mať fyzický prístup k počítaču a pri takýchto útokoch sa často získavajú dáta hlavne vyhotovením obrazu operačnej pamäte, prípadne odpočúvaním komunikácie komponentov počítača po určitých zberniciach. Softvérové útoky môže útočník vykonať aj na diaľku (napríklad podsunutím vhodne upravených vstupov) a väčšinou zneužívajú neošetrené pretečenia premenných, zraniteľnosti v návrhu softvéru ale aj zraniteľnosti v celkovom návrhu architektúry moderných počítačov. Pomocou softvérových útokov môže útočník získať vyššie privilégiá a vykonávať škodlivý kód alebo pomocou nich môžu uniknúť časti pamäte obsahujúe dáta citlivého charakteru.

2.1 Hardvérové útoky

Na prevedenie útoku **hot boot** útočník počítač najskôr reštartuje a následne sa operačný systém zavedie z útočníkovho média. Ak je na počítači povolené zavádzanie z pripojeného média, tak sa načítajú útočníkove programy na forenznú analýzu a začne sa robiť výpis pamäte. Niektoré BIOS-y sú však nastavené tak, aby pri zavádzaní premazali obsah RAM.

Pri útoku typu **cold boot** je potrebné preniesť RAM z počítača obete do počítača útočníka, kde sa následne pamäť skúma. Podľa [4] útočník obyčajne použije tekutý dusík alebo stlačený vzduch, aby „zmrazil“ RAM a tým dosiahol pomalšie vytrácanie sa dát z pamäte (keďže pamäť je volatilná, čiže jej schopnosť uchovávať informácie závisí od toho, či je napájaná). Výhodou tohto útoku je dokonalý obraz pamäte, pretože nedôjde k jej porušeniu samotným spúšťaním nástrojov na forenznú analýzu na skúmanom systéme. Normálne by PC mali pri vypnutí urobiť premazanie pamäte, lenže to je podľa [4] príliš drahá operácia a preto ju operačné systémy väčšinou nerobia.

Odpočúvanie Direct Memory Access (DMA – funkcionalita umožňujúca hardvérovým komponentom priamy prístup do pamäte) môže byť podľa [5] realizované, pretože i386 architektúra dôveruje každému zariadeniu pripojenému na Peripheral Component Interconnect zbernicu (PCI – zbernica na pripojenie periférnych zariadení k počítaču). Na DMA prístup cez PCI zbernicu sa ale používajú špeciálne nástroje,

ktoré sa dajú kúpiť, ale podľa [5] je ich dostupnosť obyčajne obmedzená iba na vojsko alebo políciu. Keďže ich dostupnosť je limitovaná, nie je to najrozšírenejší útok. Vytvorenie výpisu pamäte istý čas trvá a preto môžu vznikať takzvané race conditions, kedy je obsah operačnej pamäte, ktorej obraz chceme získať, čiastočne prepísaný programom slúžiacim na jej získavanie tohto obrazu. V [5] je popísaná aj obrana voči tomuto typu útokov – v zásade ide o to, že systém spadne, kým sa RAM stihne nakopírovať alebo poskytne útočníkovi nepravé výsledky.

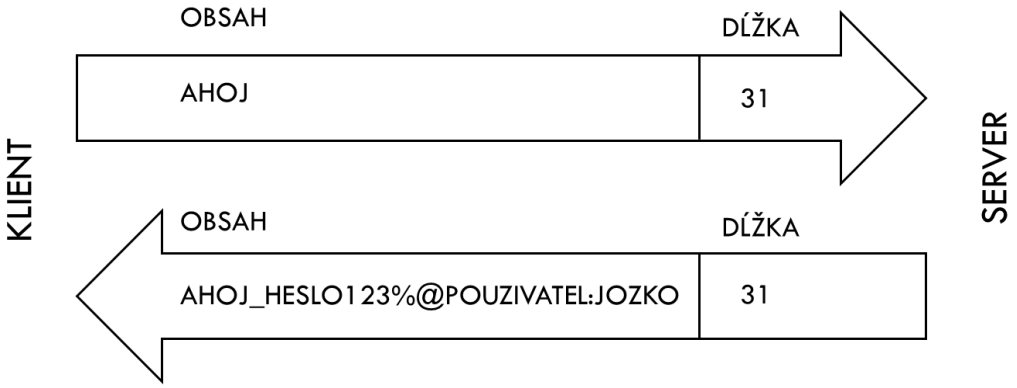
Firewire je špeciálnym prípadom DMA útoku. Má však voči nemu jednu veľkú výhodu. Firewire kábel a rovnako aj počítač s firewire prípojkou sa dajú veľmi ľahko zohnať. Pravdepodobne sa jedná o najjednoduchšiu a najlacnejšiu metódu, ktorú je možné v rámci DMA útokov použiť. Pomocou Firewire sa pamäť získava o niečo dlhšie ako cez klasický DMA útok a preto môže vzniknúť viac race conditions.

2.2 Softvérové útoky

Útok pretečením (buffer overflow) nastane, ak program nekontroluje veľkosť vstupu a útočník zadá dlhší vstup, než je časť pamäte, do ktorej sa načítava. Vtedy sa zapisovanie do pamäte neskončí na mieste vyhradenom pre danú premennú, ale pokračuje aj ďalej, mimo jeho hranice, do ďalších častí operačnej pamäte pridelenej programu. Takýmto spôsobom môže útočník prepísať hodnoty premenných, ale aj adresy inštrukcií, ktoré sa majú vykonať. Jedným z dôsledkov pretečenia môže byť podľa [6] prepísanie hesla, s ktorým pri autentifikácii testujeme zhodu a útočník sa tak môže prihlásiť aj bez znalosti správneho hesla. Ak sa pomocou pretečenia prepíše adresa kódu na vykonanie nasledujúcej inštrukcie, útočník môže získať vyššie privilégiá (keďže nový proces dedí privilégiá rodičovského) alebo vykonať nežiadúci kód [6]. Tento nežiadúci kód môže napríklad urobiť obraz operačnej pamäte a ten zaslať útočníkovi, ktorý v ňom môže hľadať citlivé údaje.

Útok **Heartbleed** je špeciálnym prípadom útoku pretečením a zneužíva rozšírenie Transport Layer Security protokolu (TLS), ktoré sa volá Heartbeat. Heartbeat zabezpečuje, že TLS relácia beží, aj keď si práve server s klientom neposielajú žiadne dáta [7]. Klient pošle serveru nejaký náhodný text spolu s jeho dĺžkou a server odpovedá rovnakým náhodným textom. Ak však útočník pošle serveru text a namiesto jeho skutočnej dĺžky pošle dĺžku oveľa väčšiu, tak server pridá do svojej odpovede toľko výplne (paddingu), aby dosiahol požadovanú dĺžku. Túto výplň pridáva z haldy. Tam sa však nachádzajú aj rôzne citlivé údaje (napríklad prihlasovacie mená, heslá

a šifrovacie kľúče) a takýmto spôsobom sa k nim útočník môže dostať. Na nasledujúcej schéme môžete vidieť, ako tento útok funguje.



Obr. 1 Schéma útoku Heartbleed

Útok **Spectre** zneužíva optimalizácie moderných procesorov. V dnešných procesoroch sa totiž kvôli zrýchleniu výpočtov používa špekulatívne vykonávanie, počas ktorého procesor vykoná aj také vetvy programu, aké by počas normálneho chodu nevykonal. Najčastejšie sa to stáva pri testovaní hraníc, kedy procesor vykoná časť kódu bez toho, aby vedel, či je splnená podmienka - či daná premenná spadá do požadovaného rozsahu. Na určenie, ktorá vetva sa má špekulatívne vykonať, sa používa algoritmus na predpovedanie nasledujúcej vykonanej vetvy. Podľa [8] ak algoritmus na predpovedanie nasledujúcej vykonanej vetvy zistí, že špekulatívne vetvy reálne nemali byť vykonané, síce budú odmietnuté a na architekturálnej úrovni sa zmeny neobjavia, ale zato zmeny na mikroarchitekturálnej úrovni ostávajú viditeľné.

Nasledujúca ukážka kódu z [8] vyobrazuje inštrukcie, ktoré sú zneužiteľné na útok typu Spectre. V kóde sa najprv vykonáva kontrola hraníc, aby sa predišlo čítaniu citlivej pamäte mimo prúdu bitov. Útočník natrénuje algoritmus na predpovedanie nasledujúcej vykonanej vetvy, aby stále vchádzal do takej vetvy, kde je premenná `x` v rámci hraníc. Urobí to tak, že niekoľkokrát za sebou vloží na vstup prípustnú hodnotu `x`. Následne vloží hodnotu, kde je `x` mimo hraníc a môže sa jednať o adresu bitu predstavujúceho časť citlivého údaju. Keď je algoritmus na predpovedanie nasledujúcej vykonanej vetvy správne natrénovaný, tak vykoná vnútro podmienkového bloku ako keby bola podmienka splnená. Zmeny sa síce na architekturálnej úrovni neprejavia – príznak ostane

nezmenený, ale zato bude v cache pamäti, čo už je zmena na mikroarchitekturálnej úrovni, ktorá nebude ihneď zvrátená.

```
if (x < bitstream_length)
    if(bitstream[x])
        flag = true
```

Na to, aby sa uniknutý bit dostal k útočníkovi, je potrebný ďalší zraniteľný kus kódu, ktorý musí používať hodnotu príznaku v ľubovoľnej operácii. Na základe zmerania času vykonania danej operácie dokáže útočník zistiť, či príznak bol v cache pamäti alebo nie. Z tejto informácie už ľahko dokáže zistiť, či hodnota v prúde bitov na pozícii `x` bola 0 alebo 1.

2.3 Získavanie citlivých údajov z obrazu pamäte

V rámci mnohých útokov, ktoré majú za cieľ únik citlivých údajov z pamäte, sa útočník nedostane priamo k citlivému údaju, ale k obrazu pamäte alebo jej časti. Z neho musí potrebné informácie správnym spôsobom vyextrahovať. Podľa [9] existujú nasledujúce prístupy k získavaniu hesiel z obrazu pamäte.

Prvým prístupom je **vytvorenie odtlačku bajtov**. Princíp tohto prístupu spočíva v tom, že sa útočník pri analýze výpisu pamäte snaží nájsť bajty, ktoré sa za každých okolností nachádzajú v okolí citlivého údaju uloženého v pamäti. Na základe získaných informácií potom vytvorí signatúru, ktorá sa dá použiť pri každom ďalšom hľadaní údaju rovnakého typu v operačnej pamäti pridelenej danej aplikácii. Príkladom takéhoto odtlačku bajtov sú nasledujúce hodnoty: 0? 00 00 00 00 75 6c 6c 00 6c 00 00 49 00 00 00. Tie sa podľa [9] stále nachádzali pred heslom aplikácie TrueCrypt v pamäti pridelenej príslušnému procesu. Znak "?" nebol vo všetkých prípadoch rovnaký, pretože reprezentoval dĺžku daného hesla.

Ďalším prístupom je **hľadanie reťazcov**. Útočník pri ňom prehľadáva priľahlú pamäť, aby našiel reťazce pravidelne sa vyskytujúce pri niektorých citlivých údajoch v pamäti.

Posledným prístupom je **znalosť umiestnenia** citlivého údaju v rámci pamäte pridelenej aplikácii. Niektoré hodnoty premememnných sa stále nachádzajú na určitých miestach v pamäti, zvlášť na určitých miestach v rámci pamäte procesu. Tieto miesta sa dajú identifikovať a použiť na hľadanie citlivých údajov.

3 Ochrana voči útokom získavajúcim dáta z RAM

Na zamedzenie útokom, ktoré majú za cieľ únik citlivých údajov z pamäte, existujú rôzne opatrenia. Mnohé riešenia sú na úrovní operačného systému a majú za cieľ skomplikovať útočníkovi získavanie citlivých údajov napríklad pomocou automatickej detekcie a prevencie pretečení premenných, pomocou zabránenia prepísaniu alebo prečítaniu častí operačnej pamäte alebo pomocou hardvérového šifrovania citlivých údajov, či dokonca šifrovania celej operačnej pamäte. Rovnako sú známe aj riešenia na alikačnej úrovni, ktoré sa snažia ochrániť citlivé údaje v rámci danej aplikácie. Ide najmä o rôzne triedy v objektovo orientovaných programovacích jazykoch, ktoré zabezpečujú bezpečné spracovanie v nich uložených citlivých údajov.

3.1 Riešenia na úrovni operačného systému

Ak raz bol napadnutý adresný priestor programu, je ťažké ochrániť dáta v ňom, pretože klasické nástroje na ochranu pamäte pracujú na úrovni procesov. Jedným z riešení je **izolácia programových modulov** (napr. ovládačov a knižníc), aby nevstupovali jeden druhému do pamäťového priestoru. Nad nedôveryhodnými modulmi môžu byť napríklad skonštruované sandboxy (bezpečnostné mechanizmy, ktoré oddeľujú bežiace procesy a obmedzujú ich prístup k zdrojom). Vtedy je výhodné ak izolované programy blízko neinteragujú. Podľa [10] ale ani takýto prístup však neochráni citlivé údaje, ak je útok prevedený v rámci toho istého modulu, kde sú uložené.

Ďalšou možnosťou je oddelenie do procesov a **vláken**, ktoré však vyžaduje veľké zmeny v architektúre programu a riešenie problémov s konkrétnym prístupom k pamäti. Opäť platí, že citlivé údaje v rámci vlákna sú zraniteľné [10]. Riešením sú **shreds**, čo je skratka pre zdieľané vlákna (shared threads) [10]. Pri vytvorení každý shred dostane časť pamäte (s-pool), ktorý môže byť zdieľaný naprieč viacerými shredmi. Shred môže pokrývať ľubovoľne veľa kódu, od jediného riadku až po celé vlákno. Nevyžaduje veľké zmeny dizajnu a vďaka vhodnému API sa jednoducho sa používa.

V [11] je popísaný **HeapTherapy** - systém, ktorý zabezpečuje ochranu voči útokom pretečením. Zahŕňa v sebe detekciu zraniteľností aj automatické generovanie záplat. Dokáže rozoznať a zabrániť prepísaniu rovnako ako aj prečítaniu citlivej pamäte a je natoľko prispôsobivý, že dokáže odhaliť aj polymorfné útoky, ktoré sa pokúšajú obísť systémy na detekciu tým, že ich kód mutuje. Na detekciu útokov pomocou pretečenia sa používajú náhodné hodnoty nazývané canaries, ktoré sa umiestnia

pred a za zraniteľnú časť pamäte a následne sa kontroluje, či boli alebo neboli poškodené, t.j. či sa ich obsah zmenil (ak nastal útok pretečením na časť pamäte, ktorá je pred nimi, tak sa ich hodnota zmenila). HeapTherapy vypočíta index pre každý zraniteľný kontext volaní vedúci k poškodeniu náhodných kontrolných hodnôt a teda k útoku pretečením. Kontext volaní je vlastne sekvencia invokovaných metód, ktoré nasledujú jedna za druhou a vedú tak k nejakej operácii. Systém pred spustením inštrukcie kontroluje jej kontext volaní, či sa náhodou nezhoduje s niektorým z indexov pre zraniteľné kontexty volaní. Ak sa zhoduje, inštrukcia sa nevykoná.

Ďalším riešením je využitie **hardvérového šifrovania**. Ukladanie kryptografických kľúčov v pamäti RAM nie je bezpečný prístup, preto sa pre tento účel vyvinulo niekoľko hardvérových riešení, ktoré sú popísané v [5] a [12]. Tie zabezpečujú, aby sa výpočty s kľúčmi vykonávali jedine v cache pamäti procesora, ktoré sú zraniteľné menej než operačná pamäť. Zároveň dokážu hardvérovo zabezpečiť aj silnú atomicitu operácií, t.j. ak sa k daným kľúčom/výpočtom pokúsi konkurentne dostať iný proces, tak to systém okamžite zistí a operácia sa preruší. Takzvaný hlavný kľúč, ktorý šifruje všetky ostatné kľúče, ak sa práve nepoužívajú, je bezpečne hardvérovo implementovaný technológiou TRESOR, čo je vlastne kryptografický stroj používajúci registre na debugovanie.

3.2 Aplikačné riešenia

Alternatívou ku `String`-u je ukladanie citlivých údajov v rámci programu do **poľa znakov**. To totiž môže byť na rozdiel od `String`-u poľahky vymazané, keď sa stane nepotrebným. Avšak stále platí, že citlivé údaje sú v tomto poli uložené nešifrovane a potenciálny útočník sa k nim vie dostať.

`SealedObject` (trieda v jazyku Java) môže byť vytvorený z ľubovoľného serializovateľného objektu. Obaľuje pôvodný objekt v serializovanej podobe a zapečatí (zašifruje) jeho serializovaný obsah. Pôvodný objekt sa dá získať dešifrovaním príslušným algoritmom a s príslušným kľúčom a následnou deserializáciou. Na získanie pôvodného objektu teda podľa [13] potrebujeme objekt `Cipher`, ktorý musí byť najprv inicializovaný so správnym algoritmom, kľúčom, výplňou atď. Výhodou tohto prístupu je, že strana, ktorá chce dešifrovať zapečatený objekt, nemusí poznať kľúč – stačí ak dostane správne inicializovaný `Cipher` objekt. Na vytvorenie `SealedObject`-u je potrebné dodať do konštruktora ľubovoľný objekt, ktorý je serializovateľný a šifru typu `Cipher`. Ak teda takýmto spôsobom chceme ochrániť obsah `String`-u, musíme

poskytnúť konštruktoru `SealedObject`-u objekty `String` a `Cipher`. To je ale kontraproduktívne, lebo práve vytvoreniu `String`-u by sme sa chceli vyhnúť. Ideálne by bolo, ak by sme vedeli reťazec načítať priamo zo vstupu a rovno ho uložiť v chránenej šifrovanej forme, bez medzikroku v podobe Stringu.

V jazyku Java existuje aj **`GuardedObject`**, ktorý na aplikačnej úrovni bráni nepovolaným procesom v prístupe k chránenému objektu. Na konštruktore je opäť chránený objekt a objekt typu `Guard`. Keďže chránený objekt nijako nešifruje a v konštuktore mu musíme poskytnúť `String`, nie je pre naše účely vhodný.

4 Analýza a návrh riešenia

V predchádzajúcich častiach sme preskúmali niekoľko útokov spôsobujúcich únik citlivých údajov z pamäte a taktiež dostupné riešenia na ochranu pred takými útokmi. Rozhodli sme sa, že v rámci nášho riešenia sa nebudeme venovať opatreniam na úrovni operačného systému, keďže programátor si počas programovania svojej aplikácie nevie vynútiť, aby operačný systém, na ktorom bude jeho aplikácia spustená, mal v sebe zahrnuté tieto opatrenia. Aplikácia môže byť napríklad spustená na serveri webhostingovej spoločnosti a programátor nemá dosah na ich operačný systém. Rovnako nevie ovplyvniť, aké bezpečnostné opatrenia bude mať v sebe zahrnuté operačný systém koncového užívateľa, ktorý si aplikáciu nainštaluje. Avšak o tom, aké triedy programátor použije vo svojom kóde, už rozhodnúť môže. Preto sa venujeme dostupným riešeniam na aplikačnej úrovni.

Preskúmali sme existujúce riešenia na aplikačnej úrovni v programovacích jazykoch Java, C# a C++. Vybrali sme si tieto jazyky, pretože sú často používané a sú dostupné pomerne rozsiahle bezpečné alternatívy k objektu typu `String` napísané práve pre tieto jazyky. Analyzovali sme nájdené bezpečné alternatívy ku `String`-u, pričom sme sa zameriavali na to, aký prístup zaujali programátori jednotlivých tried k nasledujúcim piatim problémom:

- vytvorenie triedy
- použitie kryptografických primitívov
- mazanie citlivého obsahu
- poskytnutie metód na sprístupnenie citlivého obsahu
- poskytnutie metód na úpravu citlivého obsahu

Všetky analyzované triedy riešia každý z týchto piatich bodov iným spôsobom. V rámci týchto bodov sme sa snažili nájsť to najlepšie možné riešenie, prípadne ho ďalej rozvinúť a vylepšiť. Na základe nájdených čŕt sme vytvorili pseudokód popisujúci vnútorné fungovanie metód triedy slúžiacej ako bezpečná alternatíva ku `String`-u (ďalej `SecureString`). Táto môže slúžiť ako predloha, podľa ktorej je možné vytvoriť implementáciu bezpečnej alternatívy ku `String`-u v ľubovoľnom programovacom jazyku.

4.1 Konštruktor

4.1.1 C#

Trieda `SecureString` v jazyku C# [14] má prázdny konštruktor a tiež konštruktor, ktorý má na vstupe pole nešifrovaných znakov s citlivým obsahom. Tie následne prevedie do poľa bajtov a zašifruje. Nešifrovaný vstup však nemaže, túto činnosť necháva na zodpovednosť programátorovi.

Poskytuje aj metódu `AppendChar` na pridanie nového písmena na koniec uloženého textu, ak by sme chceli napríklad tvoriť objekt typu `SecureString` zo zdroja, ktorý poskytuje nešifrovaný text písmeno po písmene. V rámci metódy `AppendChar` sa chránená časť pamäte s citlivými údajmi najskôr dešifruje, pridá sa daný znak a následne je opäť zašifrovaná. Táto časť kódu je obalená v `try-catch-finally` bloku a opätovné zašifrovanie danej časti pamäte je zabezpečené rovnako v `catch` ako aj vo `finally` vetve. To znamená, že aj keby počas behu programu došlo k neočakávanej výnimke, pamäť s citlivým obsahom nezostane nechránená.

To, že sa pri pridávaní znaku citlivý obsah dešifruje a znovu zašifruje jednak môže spôsobovať spomalenie, pokiaľ je šifrovanie výpočtovo náročné, a okrem toho daný prístup spôsobuje, že sa nešifrovaný citlivý obsah bude nachádzať v pamäti častejšie ako by sme chceli, aj keď bude následne vymazaný.

4.1.2 Java

Trieda `GuardedString` v jazyku Java [15] poskytuje takisto prázdny konštruktor aj konštruktor z poľa nešifrovaných znakov s citlivým obsahom, ktoré prevedie na pole bajtov a zašifruje. Vymazanie nešifrovaného poľa znakov opäť necháva na zodpovednosť programátorovi.

Pokiaľ sme použili prázdny konštruktor a máme zdroj nešifrovaného citlivého obsahu písmeno po písmene (napríklad pri čítaní z konzoly), vieme využiť metódu `appendChar` na prilepenie nového písmena za existujúci text. Táto metóda spôsobí, že sa nešifrovaný citlivý obsah bude na krátky čas nachádzať v pamäti dvakrát. Prvýkrát sa citlivý obsah dešifruje do poľa znakov, druhýkrát sa vytvorí o 1 väčšie pole znakov, kam sa obsah prekopíruje a pridá sa tam znak, ktorý sme chceli. Následne vo `finally` bloku dôjde k vymazaniu oboch týchto nešifrovaných polí znakov. Tento prístup predstavuje rovnaký problém ako v C#, akurát s tým rozdielom, že v jazyku Java sa nešifrovaná kópia bude dočasne nachádzať v pamäti dokonca dvakrát.

4.1.3 C++

Trieda `SecureString` v jazyku C++ [16] poskytuje:

- prázdny konštruktor
- konštruktor, ktorý má na vstupe iba dĺžku
- konštruktor, ktorý vytvorí `SecureString` iba z iného objetu typu `SecureString`
- konštruktor, ktorý má na vstupe citlivý obsah ako pole znakov, jeho dĺžku a `boolean` premenné `deleteStr` a `allowNull`

Premenná `deleteStr` je implicitne nastavená na hodnotu `true`. To znamená, že pokiaľ sa to programátor nerozhodne zakázať a nenastaví premennú `deleteStr` na `false`, pôvodné nešifrované pole znakov bude po vytvorení objektu typu `SecureString` vymazané.

Na začiatku vykonávania kódu konštruktora trieda `lock_guard` pomocou vzájomného vylúčenia (mutex) uzamkne daný blok kódu pre volajúce vlákno a po jeho vykonaní ho znovu uvoľní. Zámok `lock_guard` spôsobí, že do kritickej sekcie nebude mať prístup iné vlákno a teda až do vykonania celého kódu konštruktora k daným premenným nepristúpia iné vlákna.

4.2 Kryptografické primitívy

4.2.1 C#

V jazyku C# trieda `SecureString` šifruje citlivý obsah pomocou metódy `Win32Native.SystemFunction040`, ktorá má na vstupe časť pamäte, ktorú chceme zašifrovať, jej dĺžku a stupeň ochrany. Avšak v dokumentácii nie je špecifikované, akým spôsobom táto systémová funkcia šifrovanie vykonáva a nie sú dostupné ani voľné zdrojové kódy.

4.2.2 Java

V jazyku Java trieda `GuardedString` na šifrovanie používa náhodný objekt slúžiaci na šifrovanie (`Encryptor`), ktorý vygeneruje pomocou `EncryptorFactory.getInstance().newRandomEncryptor()`. Rovnako ako pri triede `SecureString` v jazyku C# ani v jazyku Java nie je bližšie špecifikované, o akú šifru ide. Trieda `GuardedString` dokonca programátorovi poskytuje aj metódu

setEncryptor, v ktorej si sám môže zvoliť, aký Encryptor bude použitý, čo môže znamenať zraniteľnosť, ak programátor zvolí slabú šifru.

4.2.3 C++

V jazyku C++ trieda SecureString šifruje pomocou funkcie XOR. Vygeneruje náhodný kľúč rovnakej dĺžky akú má aj pole s citlivými dátami a následne urobí XOR každého prvku poľa citlivých údajov s hodnotou kľúča na príslušnom indexe. Jednotlivé časti kľúča generuje pomocou metódy rand(). V rámci inštančných premenných sa potom v danej triede zapamätáva šifrovaný text a samotný kľúč. Bezpečnosť citlivého obsahu teda závisí od spoľahlivosti náhodného generátora a od toho, či potenciálny útočník bude môcť nájsť v pamäti kľúč a šifrovaný obsah.

Metóda rand() je nespoľahlivý pseudonáhodný generátor, preto by bolo vhodné použiť lepší pseudonáhodný generátor alebo aj úplne náhodný generátor. Podľa [17] je napríklad možné použiť metódu srand(time(0)), ktorá na generovanie náhodných čísel použije náhodný seed, ktorým je v tomto prípade aktuálny čas.

4.3 Sprístupnenie citlivého obsahu

4.3.1 C#

V jazyku C# trieda SecureString poskytuje na prácu s nešifrovaným citlivým obsahom metódy ToBSTR, ToUniStr a ToAnsiStr. Metóda ToBSTR vracia ukazovateľ na reťazec znakov ukončený nulovým znakom. Najprv dešifruje chránenú časť pamäte s citlivým obsahom, vráti daný ukazovateľ a v catch aj finally bloku zabezpečí, aby boli citlivé dáta opäť zašifrované. Ak aj došlo k nejakému zlyhaniu, tak sa časť pamäte vyprázdni a ukazovateľ na ňu bude uvoľnený. Metódy ToUniStr a ToAnsiStr vracajú ukazovateľ na reťazec v príslušnom kódovaní (Unicode alebo ANSI). Citlivý obsah je najprv dešifrovaný a do potrebného kódovania je prevedený pomocou triedy Marshaller. Opätové šifrovanie pamäte je rovnako zabezpečené v catch a finally bloku.

4.3.2 Java

V jazyku Java sa v triede GuardedString na prácu s citlivým obsahom používa rozhranie spätného volania (callback interface) nazývaný Accessor. Práca s Accessor-om a sprístupnenie citlivých údajov sú vyobrazené v nasledujúcej ukážke. Accessor je rozhranie nachádzajúce sa priamo v triede GuardedString

a má práve jednu abstraktnú metódu, ktorú je potrebné implementovať – daná metóda sa nazýva `access`. Ak niekto na objekte typu `GuardedString` zavolá metódu `access`, musí jej na vstupe poskytnúť implementáciu rozhrania `Accessor`, t.j. implementovať metódu `access` patriacu rozhraniu `Accessor`. (Metóda `access` volaná z triedy `GuardedString` a `access` ako abstraktná metóda pre rozhranie `Accessor` sú dve rôze metódy.) Trieda `GuardedString` sa sama postará o to, aby metóda `access`, ktorú pre `Accessor` implementoval programátor, dostala na vstupe pole nezašifrovaných znakov citlivého obsahu, vykonala, čo programátor zadefinoval a aby následne bolo toto nešifrované pole automaticky bez zásahu programátora vymazané.

```
public final class GuardedString {

public interface Accessor {
        public void access(char[] clearChars);
}

public void access(Accessor accessor) {
         checkNotDisposed();
         char[] clearChars = null;
         try {
             clearChars = decryptChars();
             accessor.access(clearChars);
         }  finally {
             SecurityUtil.clear(clearChars);
         }
     }
}
```

Tento prístup je dobrý kvôli tomu, že mazanie sa deje bez zásahu programátora – dôjde k nemu automaticky po vykonaní požadovanej akcie s nešifrovaným citlivým obsahom. Jediným problémom môže byť, ak si programátor v rámci impelmentácie abstraktnej metódy `access` uloží citlivý obsah do nejakej lokálnej premennej v rámci svojho kódu (mimo rozhrania `Accessor`). Vtedy už je programátor sám zodpovedný za vymazanie tejto lokálnej premennej a nezodpovedá za to trieda `GuardedString`.

4.3.3 C++

V jazyku C++ trieda `SecureString` za účelom prístupu k citlivému obsahu vracia ukazovatele na daný obsah. Metóda `getUnsecureString` vráti ukazovateľ na nešifrovanú kópiu citlivého obsahu, pričom príznak *zmeniteľný (mutable)* nastaví na false - to znamená, že obsah tejto nešifrovanej kópie nie je možné meniť. Avšak stále je možné vynulovať nešifrovaný citlivý obsah metódou `memset`, ktorá na miesto, kde sa citlivý obsah nachádza v pamäti, napíše nuly. Metóda `getUnsecureString`

vracia stále len jednu kópiu nešifrovaného citlivého obsahu, ale ak už takáto kópia nešifrovaného obsahu bola vrátená, po ďašom zavolaní metóda `getUnecureString` vráti `NULL`.

Metóda `getUnsecureStringM` vracia *zmeniteľnú (mutable)* kópiu nešifrovaného citlivého obsahu (príznak *zmeniteľný (mutable)* je nastavený na true), ale opäť zabezpečuje, že v rovnakom čase môže existovať len jedna kópia nešifrovaného citlivého obsahu – ak by sa opätovým zavolaním metódy mala vytvoriť ďalšia kópia, nestane sa tak, ale metóda vráti `NULL`. Metóda `getUnsecureStringM` vracia ukazovateľ na reťazec a v pamäti alokuje priestor takej veľkosti, akú dĺžku má daný reťazec. Ak sa teda obsah reťazca mení, môže dôjsť k poškodeniu haldy, ak nepozorný programátor urobí úpravy, ktoré majú za následok nárast dĺžky tohto reťazca.

V jazyku C++ v triede `SecureString` existuje aj metóda `getUnsecureNextLine`, ktorá vráti nezmeniteľnú (immutable) kópiu nešifrovaného citlivého obsahu až po nasledujúci znak reprezentujúci posun do ďalšieho riadku (opäť povoľuje existenciu iba jedinej takej kópie v danom čase a v opačnom prípade vracia `NULL`).

Po zavolaní metód `getUnsecureString`, `getUnsecureStringM`, prípadne `getUnsecureNextLine` a urobení požadovaných operácií na nešifrovanom citlivom obsahu musí programátor zavolať metódu `getUnsecureStringFinished`. Tá v prípade, že bola vrátená zmeniteľná kópia nešifrovaného citlivého obsahu, zašifruje túto nešifrovanú kópiu citlivého obsahu aj s vykonanými zmenami a uloží ju v rámci inštančných premenných triedy na miesto pôvodného citlivého obsahu. Ak bola vrátená nezmeniteľná kópia dešifrovaného citlivého obsahu, tak ju pomocou metódy memset vynuluje.

4.4 Metódy na úpravu citlivého obsahu

4.4.1 C#

V jazyku C# trieda `SecureString` umožňuje programátorovi zistiť dĺžku citlivého obsahu (metóda `Length`), skopírovať obsah do iného objektu typu `SecureString` (metóda `Copy`). Spomedzi skúmaných tried táto umožňuje najviac práce s citlivým obsahom, pretože dovoľuje nielen pridať znak (metóda `AppendChar`), ale aj znak z ľubovoľnej pozície odstrániť (metóda `RemoveAt`), na ľubovoľnom indexe ho zmeniť (metóda `SetAt`) a na ľubovoľný index nejaký znak vložiť (metóda `InsertAt`).

Na vykonanie týchto operácií nad citlivým obsahom je ale nutné najprv ho dešifrovať, urobiť zmeny a následne ho (vo `finally` bloku) opätovne zašifrovať.

4.4.2 Java

V jazyku Java trieda `GuardedString` umožňuje skopírovať obsah do novej inštancie objektu `GuardedString` (metóda `copy`), overiť či sa dva objekty typu `GuardedString` rovnajú (metóda `equals`) a vrátiť haš z citlivého obsahu (metóda `hashCode`). Obsahuje metódu, ktorá overí, či sa zadaný haš rovná hašu citlivého obsahu (metóda `verifyBase64SHA1Hash`). Vývojári triedy `GuardedString` sa pravdepodobne rozhodli použiť SHA1 haš z reťaca zakódovaného kódovaním base64, pretože sa jedná o jeden z najrozšírenejších spôsobov ukladania hesiel do databázy. Aj metóda `equals` na overenie rovnosti dvoch objektov typu `GuardedString` využíva metódu `verifyBase64SHA1Hash`. `GuardedString` neposkytuje programátorovi nijakú možnoť na zmenu obsahu, ktorý uchováva, presnejšie povedané nedovoľuje meniť ani mazať už uložené a zašifrované znaky. Dovoľuje jedine pridať nový znak k už existujúcemu obsahu (metóda `appendChar`). V rámci metódy `appendChar` sa ale neaktualizuje inštančná premenná `base64SHA1Hash`, čím sa hodnota tejto premennej stáva nevyužiteľnou.

4.4.3 C++

V jazyku C++ trieda `SecureString` dovoľuje programátorovi zistiť dĺžku citlivého obsahu a veľkosť miesta, ktoré je na tento obsah alokované – tieto dve hodnoty sa totiž nemusia rovnať (metódy `length` a `allocated`). Ďalej umožňuje porovnať, či sa citlivý obsah rovná citlivému obsahu iného objektu typu `SecureString`, prípadne či sa rovná obsahu nejakého poľa znakov (metóda `equals`). Tiež dovoľuje pridať na koniec citlivého obsahu nový znak (metóda `append`), ale neposkytuje už metódy na inú zmenu obsahu. Trieda umožňuje zistiť programátorovi hodnotu znaku na poskytnutom indexe (metóda `at`).

4.5 Porovnávanie obsahu (metóda equals)

V súčasnosti je väčšia pravdepodobnosť, že moderné kryptografické systémy budú prelomené skôr útokom cez bočný kanál než matematickými výpočtami [18]. V kryptografii sa takzvaný útok meraním času (timing attack) klasifikuje ako jeden z útokov cez bočný kanál (side channel attacks). Útočník pomocou neho zistí, ako vyzeral

nezašifrovaný text len pomocou merania času, ktorý prejde počas vykonávania kryptografických algoritmov.

Nasledujúca ukážka predstavuje kód na porovnanie obsahu dvoch reťazcov, ktorý je zraniteľný na útok pomocou merania času. Tento útok meraním času je možný, pretože porovnávanie objektov typu `String` je optimalizované a ihneď ako nastane nezhoda, je prerušené.

```
if EQUALS (String1, String2) then
  return true
else
  return false
end if
```

Aby porovnávanie objektov typu `String` mohlo prebehnúť, oba musia byť rovnakej dĺžky. Ak sa ich dĺžka nebude zhodovať, okamžite nastane návrat a to bez hocijakých ďalších výpočtov a porovnaní. Dĺžka objektu `String` teda môže uniknúť skrz útok meraním času takým spôsobom, že útočník postupne porovnáva `String`-y rôznej dĺžky s pôvodným reťazcom. Keď zmeria časy potrebné na porovnanie, zistí aká bola dĺžka pôvodného reťazca – ak porovnanie trvalo dlhšie, znamená to, že našiel správnu dĺžku, lebo nenastal návrat ihneď na začiatku ako je tomu pri nerovnakých dĺžkach dvoch `String`-ov.

Podobným spôsobom vie útočník odhaliť aj jednotlivé znaky nachádzajúce sa v danom objekte `String`. Pri porovnávaní totiž nastane návrat ihneď ako sa nájdu dva znaky, ktoré sú rôzne. Povedzme, že tajný obsah, ktorý sa útočník pokúša nájsť, je slovo „auto“. Útočník môže `String` obsahujúci tajný obsah postupne porovnávať s iným objektom `String`, nad ktorého obsahom má kontrolu on. Najprv ho porovná so `String`-om „aula“, potom so `String`-om „pole“. Zmeria čas potrebný na porovnanie a zistí, že porovnanie so `String`-om „aula“ trvalo dlhšie, pretože prvé písmená sa zhodovali a nebolo prerušené hneď na začiatku. Teda už vie, že prvé písmeno tajného `String`-u je „a“. Ak útočník postupne porovnáva `String`, ktorého obsah chce zistiť, s iným `String`-om, ktorého obsah vie kontrolovať, môže písmeno po písmene uhádnuť, ako vyzeral pôvodný reťazec.

Jedným zo spôsobov ako sa vyhnúť útokom meraním času je zabezpečiť, aby porovnávanie vždy trvalo rovnako dlho. V nasledujúcej ukážke inšpirovanej kódom zo [18] môžete vidieť, ako vyzerá takéto porovnávanie dvoch reťazcov, ktoré trvá

konštantne dlho. Tento kód ale žiaľ nezaručuje vykonávanie porovnávania v konštantnom čase na každej platforme a pri každom kompilátore – niekde môže byť optimalizovaný neočakávaným spôsobom.

```
function CONSTANT TIME COMPARE (a, b)
  Difference <- 0
  MinimalLength <- MINUMUM (LENGTH (a), LENGTH (b))
  for i ∈ (0, MinimalLength) do
    Difference <- a[i] XOR b[i]
  end for
  if LENGTH (a) ≠ LENGTH (b) then
    return false
  end if
  return EQUALS (Difference, 0)
end function
```

Porovnávanie `String`-ov pomocou hašov má podobný efekt – tiež zabráni útoku meraním času, pretože z porovnávania hašov nám neunikne ani informácia o dĺžke jednotlivých reťazcov ani o tom, aké znaky sa v pôvodných reťazcoch rovnajú. Je to spôsobené tým, že haše vygenerované rovnakým algoritmom majú vždy rovnakú dĺžku a to, či sa haše na danom indexe rovnajú, nie je nijako ovplyvnené faktom, či sa rovnajú aj príslušné znaky v pôvodných reťazcoch.

4.6 Vymazanie citlivého obsahu

Vymazanie citlivého obsahu musí byť zaručene vykonané – musí byť prevedené bezpečným spôsobom. Ten však už závisí od špecifík daného jazyka. Väčšinou za týmto účelom existujú rôzne špecializované funkcie zabezpečujúce bezpečné vymazanie z pamäte.

Ak totiž najprv pracujeme s poľom citlivých údajov a následne ho vpylníme nulami, kompilátor v rámci svojich optimalizácií môže zistiť, že nový obsah poľa (teda samé nuly) sa už nikde nepoužíva a preto túto operáciu vynechá a nevykoná ju [19]. Vykonávanie bude teda zoptimalizované, ale naše citlivé údaje ostanú v poli v pôvodnej nezmenenej forme.

5 Pseudokód SecureString

5.1 Konštruktor

Keď sme pri implementácii volili spôsob, akým budeme náš `SecureString` vytvárať, rozhodli sme sa použiť prázdny konštruktor a pridávanie jednotlivých znakov písmeno po písmene pomocou metódy `APPEND CHARACTER`. Tento prístup je užitočný pri načítavaní citlivého vtupu z konzoly, ak ho nechceme priebežne nikde ukladať, ale rovno z neho tvoriť šifrovaný obsah.

Ďalej sme zvolili konštruktor z poľa znakov, pričom po vytvorení `SecureString`-u a zašifrovaní citlivého obsahu naša trieda premaže pôvodný citlivý obsah v poli znakov poskytnutom na vstupe v konštruktore a nenecháva to na zodpovednosti programátora.

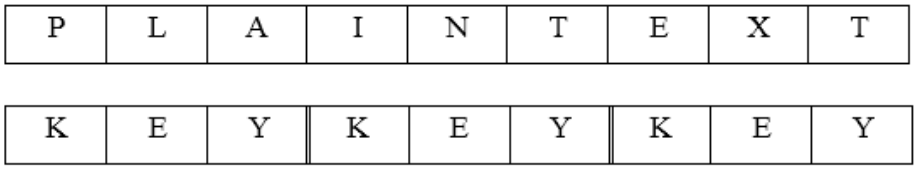
Operácie v konštruktore pri vytváraní `SecureString`-u sme obalili do `try` bloku a vo `finally` bloku sme zaistili vymazanie poľa s citlivými znakmi. Použitie `finally` bloku sme zvolili kvôli tomu, aby k zašifrovaniu došlo vždy – aj pokiaľ by v `try` bloku vznikla nejaká výnimka. Keďže konštruktor je jediné miesto, kde ešte máme prístup k nešifrovanej reprezentácii pôvodného citlivého obsahu, tak v rámci konštruktora vytvárame z tohto obsahu haše, ktoré môžu byť neskôr použité pri porovnávaní, či sa dva `SecureString`-y rovnajú. V nasledujúcej ukážke môžete vidieť časť pseudokódu, ktorá zabezpečuje vyššie popísaný postup.

```
try
  SHA2Hash <- SHA2 HASH (PlaintextCharacters)
  SHA3Hash <- SHA3 HASH (PlaintextCharacters)
  WhirlpoolHash <- WHIRLPOOL HASH (PlaintextCharacters)
  ENCRYPT CHARACTERS (PlaintextCharacters)
end try
finally
  for all Character ∈ PlaintextCharacters do
    Character <- 0
  end for
end finally
```

5.2 Kryptografické primitívy

V našej implementácii sme sa rozhodli pre šifrovanie pomocou funkcie `XOR`, ktorá má na vstupe nešifrovaný citlivý obsah a kľúč. Kvôli bezpečnosti používame kľúč rovnakej dĺžky akú má aj citlivý obsah. Ak by bol kľúč kratší ako citlivý obsah (t.j. kľúč by sme opakovali toľkokrát dookola až pokým by sme nedosiahli dĺžku citlivého obsahu – viď. Obr. 2), šifra by sa dala prelomiť frekvenčnou analýzou [20]. Ak je ale kľúč

taký dlhý ako citlivý obsah (t.j. nemusíme ho opakovať) a znaky v ňom sú dostatočne náhodné, prelomenie frekvenčnou analýzou pre nás nepredstavuje riziko.



Obr. 2. Nebezpečné šifrovanie pomocou funkcie XOR s krátkym kľúčom

Šifrovací kľúč generujeme cez generátor náhodných čísle. Ak by sme použili nespoľahlivý pseudonáhodný generátor a útočník by dokázal predvídať alebo vypočítať, aké náhodné čísla budú vygenerované a teda ako bude vytvorený kľúč, dokázal by poľahky dešifrovať aj citlivé údaje.

Aby útočníkovi na získanie citlivého obsahu nestačilo v obraze pamäte hľadať dva reťazce rovnakej dĺžky (kľúč a šifrovaný text), rozhodli sme sa generovať kľúč väčšej dĺžky ako je dĺžka citlivého obsahu. Z takéhoto kľúča potom do funkcie `XOR` s citlivým obsahom použijeme iba časť (samozrejme časť rovnakej dĺžky, akú má aj citlivý obsah). Dĺžku kľúča si vždy vygenerujeme ako náhodný násobok dĺžky citlivého obsahu. Použitie dlhšieho kľúča v našom prípade znamená, že v rámci inštančných premenných triedy si potrebujeme zapamätať šifrovaný text, kľúč a index, od ktorého daný kľúč reálne používame (offset). Tieto tri premenné potrebujeme na dešifrovanie. V nasledujúcej ukážke môžete vidieť generovanie takéhoto kľúča.

```
Ratio <- TRNG (2.25, 9.25)
KeyLength <- PlaintextLength * Ratio
KeyOffset <- TRNG (0, KeyLength - PlaintextLength)
for i ∈ (0, KeyLength) do
  Key[i] <- TRNG ()
end for
```

V ďalšej ukážke zase môžete vidieť metódu na dešifrovanie. Táto metóda však nie je sprístupnená „zvonku“, programátor nikdy priamo nedostane pole dešifrovaných znakov. Metóda je použitá iba interne v rámci metódy `ACCESS`. Môžeme vidieť, že celé šifrovanie spočíva iba v použití funkcie `XOR` na citlivý obsah a kľúč (od určitého indexu). Vo `finally` bloku je opäť zabezpečené vymazanie poľa bajtov, v ktorom sa nachádza nešifrovaný citlivý obsah.

```
try
  for i ∈ (0, LENGTH (Ciphertext)) do
    PlaintextBytes[i] <- Ciphertext[i] XOR Key[KeyOffset + i]
  end for
  return TO CHARACTER ARRAY (PlaintextBytes)
end try
finally
  for all Byte ∈ PlaintextBytes do
    Byte <- 0
  end for
end finally
```

5.3 Sprístupnenie citlivého obsahu

Prístup v jazyku C# rovnako ako aj v jazyku C++ spočíva v tom, že programátor dostane referenciu na nešifrovaný obsah. To má za následok, že programátor musí citlivý obsah po vykonaní potrebných operácií vymazať, nepostará sa o to príslušná trieda. Preto sme sa rozhodli tento prístup v rámci našej implementácie nepoužiť.

V rámci našej implementácie sme sa rozhodli zvoliť si rovnaký prístup, aký zvolili vývojári triedy `GuardedString` v jazyku Java a tým je použitie rozhrania spätného volania (v triede `GuardedString` sa nazýva `Accessor`). Pri tomto prístupe je mazanie citlivých údajov čo najviac automatizované a primárne zaňho zodpovedá trieda na uchovávanie citlivých údajov a nie programátor.

Naša trieda `SecureString` teda na rovnakom princípe pomocou rozhrania spätného volania zabezpečuje vymazanie citlivého obsahu v podobe poľa nešifrovaných znakov z pamäte ihneď po jeho spracovaní. Podrobne viď. Príloha A, riadky 84-100. Ak si programátor v implementácii metódy `ACCESS` explicitne neuloží citlivý obsah z poskytnutého poľa nešifrovaných znakov do inej premennej, tak citlivý obsah ostane chránený. Ak by si citlivé údaje uložil do inej premennej, bol by zodpovedný za jej vymazanie.

5.4 Metódy na úpravu citlivého obsahu

Metódu `CHARACTER AT` sme sa rozhodli neimplementovať, pretože na prácu s nešifrovaným citlivým obsahom používame `ACCESSOR`, ktorý následne vymaže všetky jemu dostupné nešifrované reprezentácie citlivého obsahu. Pri metóde `CHARACTER AT` by sme nemali takúto kontrolu nad znakmi citlivého obsahu.

Rozhodli sme sa ale implementovať metódy `SET CHARACTER AT`, `INSERT CHARACTER AT` a `REMOVE CHARACTER AT`. Tie umožnia dodatočne upravovať

citlivý obsah našej triedy a to konkrétne zmeniť znak na danom indexe, vložiť znak za daný index a vymazať znak na určenom indexe.

Keďže sme na zašifrovanie obsahu použili XOR s kľúčom rovnakej dĺžky ako je dĺžka citlivého obsahu, nie je potrebné v rámci týchto metód dešifrovať celý citlivý obsah, vykonať dané operácie a následne ho zašifrovať – stačí pracovať s hodnotami na daných indexoch. Obsah vlastne dešifrovať ani vôbec nemusíme, dokonca ani na príslušných indexoch.

Pri metóde INSERT CHARACTER AT stačí, ak zväčšíme pole s uloženým kľúčom a na vzniknuté prázdne miesta si vygenerujeme toľko náhodných bajtov, koľko znakov chceme vložiť. Rovnako si zväčšíme aj pole s uloženým šifrovaným textom a na daný index v šifrovanom texte zapíšeme výsledok funkcie XOR z požadovaného znaku a kľúča na príslušnom indexe. V nasledujúcej ukážke môžete vidieť ako rozširujeme pole so šifrovaným citlivým obsahom a vkladáme doňho nové zašifrované znaky (za predpokladu, že kľúč sme si už predom rozšírili o potrebný počet náhodných bajtov). Nakoniec pôvodné pole so šifrovaným textom vymažeme, keďže teraz sme si už jeho obsah nakopírovali do nového, väčšieho poľa a pôvodný obsah nemáme dôvod naďalej uchovávať.

```
for i ∈ (0, Position) do
  NewCiphertext[i] <- Ciphertext[i]
end for
for i ∈ (0, LENGTH (Characters)) do
  NewCiphertext[Position + i] <- Characters[i] XOR Key[KeyOffset +
Position + i]
end for
for i ∈ (Position, LENGTH (Ciphertext)) do
  NewCiphertext[i + LENGTH (Characters)] <- Ciphertext[i]
end for
for i ∈ (0, LENGTH (Ciphertext)) do
  Ciphertext[i] <- 0
end for
Ciphertext <- NewCiphertext
```

V rámci metódy SET CHARACTER AT vo funkcii XOR nemôžeme s novým znakom použiť rovnakú hodnotu kľúča na danom indexe, aká tam bola predtým. Ak totiž rôzne citlivé obsahy zašifrujeme rovnakými kľúčmi, umožnilo by to rôzne útoky a následné odhalenie kľúča. Preto musíme na danom indexe vygenerovať novú časť kľúča a až tú použiť vo funkcii XOR s novými znakmi, ktorými chceme nahradiť pôvodné. V nasledujúcej ukážke môžete vidieť časť pseudokódu, kde sa generuje nová časť kľúča,

ktorou sa potom šifruje citlivý obsah, ktorým nahrádzame predchádzajúci obsah na daných indexoch.

```
for i ∈ (0, LENGTH (Characters)) do
  Key[KeyOffset + Position + i] <- TRNG ()
end for
for i ∈ (0, LENGTH (Characters)) do
  Ciphertext[Position + i] <- Key[KeyOffset + Position + i] XOR
Characters[i]
end for
```

Pri metóde `REMOVE CHARACTER AT` je postup najjednoduchší – stačí skrátiť kľúč a pole obsahujúce šifrovaný text a odstrániť prvok na danom indexe.

5.5 Porovnávanie obsahu (metóda equals)

V jazyku Java si trieda `GuardedString` ukladá SHA1 haš citlivého obsahu zakódovaného pomocou kódovania base64. Ten ale už nie je považovaný za bezpečný. Naša trieda `SecureString` si ponecháva uložené SHA2, SHA3 a Whirlpool haše pôvodného citlivého obsahu. Vybrali sme práve tieto haše, pretože sú odporúčané ako bezpečné podľa ENISA štandardov popísaných v [21].

Je výhodné mať uložené haše citlivého obsahu, pretože sa dajú využiť napríklad pri kontrole, či sa daný haš hesla z databázy zhoduje s obsahom našej triedy `SecureString`. Takisto je vhodné použiť pri zisťovaní, či sa dva objekty typu `SecureString` zhodujú, práve porovnávanie cez haš. Tento prístup nám zabráni rôznym útokom meraním času spomínaným v kapitole 4.5.

6 Splnenie požiadaviek PCI DSS štandardu

Payment Card Industry Data Security Standard (PCI DSS) alebo Štandard pre dátovú bezpečnosť v oblasti platobných kariet je štandard, ktorý bol podľa [22] navrhnutý, aby zabezpečil širokospektrálne, konzistentné a globálne opatrenia chrániace dáta z platobných kariet. Poskytuje technické a procesné požiadavky, ktoré boli spísané, aby chránili citlivé údaje súvisiace s platobnými kartami a bankovými prevodmi, a ktoré sú povinné dodržiavať všetky entity spracúvajúce, ukladajúce alebo preposielajúce údaje viažúce sa k platobnej karte, k jej držiteľovi alebo autentifikačné údaje. Tento štandard obsahuje 12 požiadaviek, pričom mnohé z nich sú procesné a hovoria napríklad o fyzickej, sieťovej bezpečnosti, monitorovaní systémov a školení zamestnancov. Našej práce sa primárne týka požiadavka č. 3, ktorá sa zaoberá ochranou uložených citlivých dát.

V tejto kapitole skúmame kompatibilitu navrhnutého pseudokódu `SecureString` s PCI DSS štandardom. Zvolili sme si tento štandard, pretože náš návrh triedy, ktorá bezpečne ukladá a spracúva citlivé údaje má šancu byť využiteľný práve pri programovaní aplikácií v odvetví týkajúcom sa bankovníctva, platobných kariet a bankových prevodov. Na to, aby boli programátori oprávnení použiť náš návrh pri vytváraní aplikácie spracúvajúcej citlivé údaje súvisiace s platobnými kartami, musí byť tento návrh kompatibilný práve s PCI DSS štandardom.

6.1 Aplikovateľné požiadavky

V požiadavke 3.1 štandardu je povedané, že uchovávanie citlivých údajov by malo byť limitované na minimálny potrebný čas a mali by existovať buď manuálne alebo automatizované procesy na nájdenie a bezpečné odstránenie nepotrebných dát. Musí sa jednať o metódy, ktoré poskytnú bezpečné vymazanie citlivých údajov bez možnosti ich spätnej rekonštrukcie. Táto požiadavka sa vzťahuje najmä na dlhodobo ukladané dáta v databázach a na rôznych úložiskách, ale je rovnakým dielom aplikovateľná aj na prechodne uložené dáta v operačnej pamäti, ktoré program práve načítava alebo spracúva, čo je práve prípad našej triedy `SecureString`.

Náš návrh triedy `SecureString` spĺňa požiadavku 3.1, pretože poskytuje metódu `CLEAR` na bezpečné vymazanie poľa so zašifrovaným citlivým obsahom rovnako ako poľa so šifrovacím kľúčom, hocikedy programátor usúdi, že citlivé údaje načítané v premennej už nie je potrebné uchovávať a môžu byť odstránené z danej premennej

aj z operačnej pamäte (viď. Príloha A, riadky 278 - 287). Ďalej naša trieda zabezpečí automatické vymazanie nešifrovaného citlivého obsahu v momente, kedy ho nie je ďalej potrebné uchovávať – konkrétne pri vytvorení objektu danej triedy okamžite po zašifrovaní citlivého obsahu (viď Príloha A, riadky 22 - 27 a 36 - 41) a tiež v metóde `ACCESS` po dešifrovaní citlivého obsahu a jeho sprístupnení programátorovi na najkratší možný potrebný okamih (viď Príloha A, riadky 94 - 99).

Požiadavka 8.2.1 určuje, že všetky prihlasovacie údaje majú byť chránené silnou kryptografiou počas prenosu aj uchovávania, čo znamená, že sa má jednať o testovaný a všeobecne akceptovaný kryptografický algoritmus so silnými kľúčmi a nie o vlastný návrh kryptografického algoritmu, ktorý môže s veľkou pravdepodobnosťou byť nespoľahlivý a relatívne ľahko prelomiteľný. Odporúča sa zvoliť si algoritmy z rôznych štandardov ako napríklad NIST SP 800-52, SP 800-57 alebo OWASP. Požiadavka 3.4 sa týka čísel Personal Account Number (PAN), ktoré majú byť uchovávané v nečitateľnej podobe, ale jednotlivé odporúčania na bezpečné ukladanie tohto údaju sú aplikovateľné aj na iné citlivé údaje. V požiadavke sa odporúča ukladať citlivé údaje v nečitatľnej podobe napríklad pomocou jednosmerných hašov (ak nie je potrebné mať spätne prístup k pôvodnému číslu), silnej kryptografie a s ňou súvisiacimi procesmi na správu kľúčov alebo pomocou šifry nazývanej one-time pad (OTP). OTP je systém, ktorý používa náhodne vygenerovaný súkromný kľúč na zašifrovanie jedinej správy, ktorá je potom dešifrovaná príslušným kľúčom. Podstata tohto prístupu spočíva vo fakte, že rovnakým kľúčom nie sú zašifrované viaceré správy, ale iba jedna. Požiadavky na bezpečný kľúč pri šifre OTP sú nasledujúce:

- kľúč musí byť úplne náhodný
- kľúč musí byť aspoň taký dlhý ako citlivý obsah
- kľúč nesmie byť nikdy znovu použitý (ani žiadna jeho časť)
- kľúč musí byť tajný

Podľa [23] ak sú tieto podmienky dodržané, potom je nemožné dešifrovať citlivý obsah alebo prelomiť šifru. Pri OTP sa na šifrovanie používa funkcia `XOR`, čo vychádza z Vernamovej šifry. Tá ale podľa [20] nebola bezpečná, pretože po využití všetkých znakov kľúča bol kľúč znovu použitý, čo umožnilo kryptoanalýzu a následné prelomenie šifry.

Trieda `SecureString` v rámci nášho návrhu ukladá haš z citlivého obsahu – konkrétne sa jedná o haše SHA2, SHA3 a Whirlpool, pri ktorých nepoužívame náhodný vstup do hašovacej funkcie a ktoré sú odporúčané v ENISA štandarde [21]. Ukladanie týchto hašov citlivého obsahu teda spĺňa požiadavku 3.4. Trieda `SecureString` zahŕňa metódy sprístupňujúce tieto haše programátorovi (viď. Príloha A, riadky 266 - 276) – ten ich môže napríklad ukladať do databázy, porovnávať ich s iným hašom alebo vykonať inú činnosť. V našom pseudokóde sa na ochranu citlivého obsahu používa OTP – na šifrovanie je použitý náhodne generovaný kľúč dĺžky väčšej ako je dĺžka citlivého obsahu (viď Príloha A, riadky 44 - 60). Kľúč nie je nikde opätovne použitý, vždy sa pomocou neho zašifruje iba jeden citlivý obsah. Ak aj používame metódy na úpravu citlivého obsahu, napríklad ak chceme zmeniť určité znaky v ňom, na zašifrovanie nových znakov už nepoužijeme pôvodný kľúč, ale na indexoch, kde nastáva zmena, generujeme novú časť kľúča (viď. Príloha A, riadky 206 - 212). Ak by ale vývojár chcel počas implementácie nášho návrhu triedy `SecureString` v niektorom programovacom jazyku použiť namiesto OTP iný kryptograficky silný algoritmus, je potrebné zmeniť metódy na zašifrovanie a dešifrovanie a niektoré z metód na úpravu obsahu (viď. Príloha A, riadky 44 – 60, 76 – 82, 121 – 141, 206 - 212).

Podmienkou pre bezpečné použitie OTP a zároveň požiadavkou 3.5 je, aby boli utajené a dostatočne ochránené aj kľúče šifrujúce citlivé údaje. To znamená, že kľúče šifrujúce kľúce by mali byť aspoň také silné ako kľúče šifrujúce údaje a mali by byť uložené na oddelených systémoch a na čo najmenej miestach. V našom prípade by ale použitie kľúča šifrujúceho kľúč bezpečnosť nijako nezvýšilo, pretože by kvôli použiteľnosti triedy musel byť tiež uložený v operačnej pamäti, rovnako kľúč šifrujúci dáta. Bolo by teda treba ochrániť aj kľúč šifrujúci kľúč a to by viedlo k nekonečnému cyklu. V našom riešení šifrovací kľúč chránime tým, že ho generujeme s väčšou dĺžkou, akú ma citlivý obsah a útočníkovi tak nestačí hľadať v pamäti dva reťazce rovnakej dĺžky. Keďže kľúč má náhodné hodnoty, nie je možné ho v pamäti odlíšiť od iného obsahu. Charakter problému, ktorý riešime v našej práci (ukladanie citlivých dát v operačnej pamäti) nám neumožňuje ukladať šifrovacie kľúče na osobitné úložisko alebo ich šifrovať pomocou hardvérového modulu – taký prístup by totiž neúmerne znížil použiteľnosť nášho riešenia, spôsobil by spomalenie, zvýšil pamäťové nároky, prípadne by zúžil použiteľnosť riešenia len na počítače vybavené hardvérovými šifrovacími modulmi a podobne. Už samotný fakt, že citlivé údaje sa väčšinu času

nenachádzajú v operačnej pamäti v nešifrovanej podobe a je možné vynútiť si ich odstránenie z operačnej pamäte, predstavuje uspokojivé riešenie problému.

Podľa požiadaviek 3.5.1 a 3.6 je podmienkou, aby existovala dokumentácia kryptografickej architektúry, detailov algoritmov, kľúčov, ich sily a trvanlivosti, aby bolo možné udržať krok s vyvíjajúcimi sa hrozbami, naplánovať aktualizácie a vedieť, aké silné kľúče používať. Táto požiadavka je tiež splnená, keďže ako dokumentácia k nášmu riešeniu `SecureString` slúži práve táto práca.

Po preskúmaní požiadaviek teda môžeme prehlásiť, že naše riešenie spĺňa požiadavky štandardu PCI DSS. Ostatné požiadavky už s bezpečným ukladaním citlivých údajov nesúvisia, respektíve sú mimo rozsahu toho, čo by mala zabezpečovať naša trieda `SecureString`.

7 Záver

Citlivé údaje používajúce sa v rôznych programoch je často najvhodnejšie spracovávať v podobe reťazcov. Mnohokrát sa preto stáva, že programátori tieto citlivé údaje ukladajú do premenných typu `String`. Tento prístup však nie je bezpečný, pretože `String` sa v operačnej pamäti uchováva v nešifrovanej podobe a preto ak sa útočník dostane k časti operačnej pamäte alebo k celému jej obrazu, môže si ľahko tieto citlivé údaje prečítať. Navyše v objektovo orientovaných programovacích jazykoch s manažovanou pamäťou si programátor nie je schopný vynútiť vymazanie objektu typu `String`, aj keď už jeho obsah bol spracovaný a nie je potrebné ďalej ho uchovávať – za vymazanie daného objektu zodpovedá `GarbageCollector`, ktorý sa správa veľmi nepredvídateľne a preto je ťažké predpovedať kedy a či vôbec bude `String` s citlivými údajmi vymazaný.

V práci sme preskúmali niekoľko útokov spôsobujúcich únik citlivých údajov z pamäte. Jedná sa buď o útoky hardvérového typu, pri ktorých musí mať útočník fyzický prístup k počítaču a niekedy musí mať k dispozícii aj špecializované nástroje, alebo o softvérové útoky, ktoré môžu byť vykonané na diaľku a zneužívajú zraniteľnosti programov alebo architektúry moderných počítačov.

Útoky na získanie dát z pamäte bývajú ošetrené rôznymi opatreniami buď na úrovni operačného systému alebo na aplikačnej úrovni. Opatrenia na úrovni operačného systému sa napríklad snažia detegovať pretečenia premenných, včas zastaviť pokusy o prepísanie alebo prečítanie citlivej časti pamäte, či poskytnúť ochranu citlivých údajov pomocou hardvérového šifrovania. Opatrenia na aplikačnej úrovni sa zase snažia poskytnúť triedy, ktoré bezpečne pracujú s citlivými údajmi a dokážu ich vhodne ochrániť, napríklad údaje šifrovať alebo nedovoliť iným vláknam alebo aplikáciám pristupovať k údajom.

V rámci nášho riešenia sme sa venovali opatreniam na aplikačnej úrovni, keďže programátor si nevie vynútiť, aby jeho aplikácia bola spúšťaná výlučne na zariadeniach, ktorých operačný systém má implementované vyššie spomínané opatrenia. Programátor dokáže ovplyvniť iba to, do akej premennej, respektíve do akého objektu uloží citlivý údaj vo svojej aplikácii. Veľa tried na bezpečné ukladanie citlivých údajov existujúcich v rôznych programovacích jazykoch však neposkytuje uspokojivú a postačujúcu ochranu, či funkcionalitu na prácu s obsahom a z tohto dôvodu sme vypracovali pseudokód, ktorý popisuje štruktúru a fungovanie triedy `SecureString` slúžiacej ako bezpečná alternatíva ku `String`-u. Tento pseudokód vznikol na základe analýzy

kladov a záporov už existujúcich implementácií bezpečnej alternatívy ku `String`-u v rôznych často používaných programovacích jazykoch (konkrétne sa jednalo o jazyky Java, C# a C++). `SecureString` v prvom rade šifruje citlivé údaje, ktoré uchováva. Taktiež poskytuje metódy na ich úpravu a porovnávanie a dovoľuje programátorovi úplne odstrániť citlivý obsah z operačnej pamäte kedykoľvek prestane byť potrebný. `SecureString` dovolí programátorovi pracovať s jeho citlivým obsahom, ale následne automaticky zabezpečí vymazanie všetkých nešifrovaných reprezentácií tohto obsahu.

Po vytvorení pseudokódu triedy `SecureString` sme kontrolovali jeho kompatibilitu s požiadavkami Payment Card Industry Data Security Standard (PCI DSS) štandardu. Tento štandard sú povinné dodržiavať a spĺňať všetky entity spracúvajúce, ukladajúce alebo preposielajúce údaje viažúce sa k platobnej karte, k jej držiteľovi alebo autentifikačné údaje. Na to, aby boli programátori oprávnení použiť implementáciu nášho návrhu pri vytváraní aplikácie spracúvajúcej citlivé údaje súvisiace s platobnými kartami, musí tento návrh spĺňať požiadavky PCI DSS štandardu.

Pri opatreniach ako je ukladanie citlivých údajov v reťazci typu `SecureString` môže nastať problém s kompatibilitou, kedy nejaká aplikácia závislá na tej našej očakáva na vstupe práve `String`. Aj napriek tomu má zmysel v rámci našej aplikácie pracovať s bezpečnejšou alternatívou a na obyčajný nezabezpečený `String` ju prekonvertovať až keď to bude nevyhnutné pri odovzdávaní na vstup ďalšej aplikácii. Značne sa tým zníži čas, kedy je citlivý obsah nezašifrovaný/nezabezpečený.

Pseudokód `SecureString` je možné použiť ako predlohu na vytvorenie implementácie bezpečnej alternatívy ku `String`-u v ľubovoľnom proramovacom jazyku, rovnako ho je možné použiť aj na vylepšenie už existujúcich implementácií. Tieto implementácie sú použiteľné na dočasné ukladanie citlivých údajov pri programovaní rozličných aplikácií. Je možné pomocou nich ukladať a spracúvať prihlasovacie údaje, autorizačné tokeny, súbory cookies, čísla občianskych preukazov, pasov a keďže návrh spĺňa aj PCI DSS štandard, môžu sa pomocou týchto implementácií ukladať aj rozličné údaje súvisiace s platobnými kartami. Na základe práce je možné upraviť už existujúce implementácie bezpečnej alternatívy ku `String`-u v rôznych programovacích jazykoch. Hodná preskúmania je aj využiteľnosť nášho návrhu v databázach nevyužívajúcich diskové úložisko, ale nevolatilnú verziu operačnej pamäte RAM (takzvané in-memory databázy).

Zoznam použitej literatúry

1. FARKAS, Shawn. Making Strings More Secure. .NET Security Blog [online]. [Navštívené 1.5.2019]. Dostupné na: https://blogs.msdn.microsoft.com/shawnfa/2004/05/27/making-strings-more-secure/
2. Fundamentals of garbage collection. Microsoft Docs [online]. [Navštívené 1.5.2019]. Dostupné na: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals
3. How to make object eligible for garbage collection in Java? GeeksforGeeks [online]. 30.5.2018. [Navštívené 1.5.2019]. Dostupné na: https://www.geeksforgeeks.org/how-to-make-object-eligible-for-garbage-collection/
4. KARAYIANNI, Stavroula a KATOS, Vasilios. Practical Password Harvesting from Volatile Memory. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering Global Security, Safety and Sustainability & e-Democracy. 2012. s. 17–22. DOI 10.1007/978-3-642-33448-1_3.
5. GUAN, Le, LIN, Jingqiang, LUO, Bo, JING, Jiwu a WANG, Jing. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. 2015 IEEE Symposium on Security and Privacy. 2015. DOI 10.1109/sp.2015.8.
6. DONALDSON, Mark. Inside the Buffer OverflowAttack: Mechanism, Method, Prevention. SANS Institute Reading Room [online]. [Navštívené 1.5.2019]. Dostupné na: https://www.sans.org/reading-room/whitepapers/securecode/buffer-overflow-attack-mechanism-method-prevention-386
7. "Heartbleed" OpenSSL Vulnerability 10 April 2014 - US-CERT. [online]. [Navštívené 1.5.2019]. Dostupné na: https://www.us-cert.gov/sites/default/files/publications/Heartbleed OpenSSL Vulnerability_0.pdf
8. SCHWARZ, Machael, SCHWARTZL, Martin, LIPP, Moritz a GRUSS, Daniel. NetSpectre: Read Arbitrary Memory over Network - gruss.cc. gruss [online]. [Navštívené 1.5.2019]. Dostupné na: https://gruss.cc/files/netspectre.pdf
9. DAVIDOFF, Sherri. Cleartext Passwords in Linux Memory. [online]. [Navštívené 1.5.2019]. Dostupné na: http://www.foo.be/cours/mssi-20072008/davidoff-clearmem-linux.pdf
10. CHEN, Yaohui, REYMONDJOHNSON, Sebassujeen, SUN, Zhichuang a LU, Long. Shreds: Fine-Grained Execution Units with Private Memory. 2016 IEEE Symposium on Security and Privacy (SP). 2016. DOI 10.1109/sp.2016.12.

11. ZENG, Qiang, ZHAO, Mingyi a LIU, Peng. HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows. 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2015. DOI 10.1109/dsn.2015.54.
12. COLP, Patrick, ZHANG, Jiawen, GLEESON, James, SUNEJA, Sahil, LARA, Eyal De, RAJ, Himanshu, SAROIU, Stefan a WOLMAN, Alec. Protecting Data on Smartphones and Tablets from Memory Attacks. ACM SIGPLAN Notices. 2015. Vol. 50, no. 4 s. 177–189. DOI 10.1145/2775054.2694380.
13. GONG, Li a SCHEMERS, Roland. Signing, Sealing, and Guarding Java™ Objects. Lecture Notes in Computer Science Mobile Agents and Security. 1998. s. 206–216. DOI 10.1007/3-540-68671-1_11.
14. Dokumentácia SecureString.cs. Reference Source [online]. [Navštívené 1.5.2019]. Dostupné na: https://referencesource.microsoft.com/#mscorlib/system/security/securestring.cs,77d68ea938f47705
15. Dokumentácia GuardedString.java. GitHub [online]. [Navštívené 1.5.2019]. Dostupné na: https://github.com/Evolveum/openicf/blob/master/framework/java/connector-framework/src/main/java/org/identityconnectors/common/security/GuardedString.java
16. CAELUS, Alex. Dokumentácia SecureString.cpp. GitHub [online]. [Navštívené 1.5.2019]. Dostupné na: https://github.com/alex-caelus/SecureString/blob/master/SecureString.cpp
17. How do I generate true random binary numbers in C++?. Quora [online]. [Navštívené 1.5.2019]. Dostupné na: https://www.quora.com/How-do-I-generate-true-random-binary-numbers-in-C++?
18. Simple string comparisons not secure against timing attacks. Information Security Stack Exchange [online]. [Navštívené 5.5.2019]. Dostupné na: https://security.stackexchange.com/questions/83660/simple-string-comparisons-not-secure-against-timing-attacks
19. How-to ensure that compiler optimizations don't introduce a security risk? Stack Overflow [online]. [Navštívené 5.5.2019]. Dostupné na: https://stackoverflow.com/questions/3785366/how-to-ensure-that-compiler-optimizations-dont-introduce-a-security-risk

20. KAHN, David. The codebreakers: the story of secret writing. New York : Macmillan Publishing Company, 1996.
21. Algorithms, Key Sizes and Parameters Report - ENISA. [online]. [Navštívené 1.5.2019]. Dostupné na: https://www.enisa.europa.eu/publications/algorithms-key-sizes-and-parameters-report/at_download/fullReport
22. PCI Security Standards Council [online]. [Navštívené 5.5.2019]. Dostupné na: https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf
23. One-Time Pad. One-Time Pad (OTP) [online]. [Navštívené 5.5.2019]. Dostupné na: https://web.archive.org/web/20140314175211/http://www.cryptomuseum.com/crypto/otp.htm
24. FAHRNBERGER, Günter. A Detailed View on SecureString 3.0. Advances in Computing Applications. 2016. s. 97–121. DOI 10.1007/978-981-10-2630-0_7.
25. FAHRNBERGER, Günter. A Second View on SecureString 2.0. Distributed Computing and Internet Technology Lecture Notes in Computer Science. 2014. s. 239–250. DOI 10.1007/978-3-319-04483-5_25.
26. ASAD, Ali a ALI, Hamza. Working with Cryptography. The C# Programmer's Study Guide (MCSD). 2017. s. 347–364. DOI 10.1007/978-1-4842-2860-9_13.
27. BOUFFARD, Guillaume, LACKNER, Michael, LANET, Jean-Louis a LOINIG, Johannes. Heap Hop! Heap Is Also Vulnerable. Smart Card Research and Advanced Applications Lecture Notes in Computer Science. 2015. s. 18–31. DOI 10.1007/978-3-319-16763-3_2.
28. FARLEY, Ryan a WANG, Xinyuan. CodeXt: Automatic Extraction of Obfuscated Attack Code from Memory Dump. Lecture Notes in Computer Science Information Security. 2014. s. 502–514. DOI 10.1007/978-3-319-13257-0_32.
29. FOCARDI, Riccardo, PALMARINI, Francesco, SQUARCINA, Marco, STEEL, Graham a TEMPESTA, Mauro. Mind Your Keys? A Security Evaluation of Java Keystores. Proceedings 2018 Network and Distributed System Security Symposium. 2018. DOI 10.14722/ndss.2018.23083.
30. KIM, Kyungtae a PYO, Changwoo. Securing heap memory by data pointer encoding. Future Generation Computer Systems. 2012. Vol. 28, no. 8 s. 1252–1257. DOI 10.1016/j.future.2011.02.006.
31. GENTRY, Craig a HALEVI, Shai. Implementing Gentry's Fully-Homomorphic Encryption Scheme. Advances in Cryptology – EUROCRYPT 2011 Lecture Notes in Computer Science. 2011. s. 129–148. DOI 10.1007/978-3-642-20465-4_9.

32. ZHAO, Qian a CAO, Tianjie. Collecting Sensitive Information from Windows Physical Memory. Journal of Computers. 2009. Vol. 4, no. 1. DOI 10.4304/jcp.4.1.3-10.

Prílohy

Priloha A: Pseudokód pre triedu SecureString

Príloha A

```
1   class SecureString
2
3     Key
4     KeyOffset
5     Ciphertext
6     SHA2Hash
7     SHA3Hash
8     WhirlpoolHash
9
10    function CONSTRUCTOR
11      CONSTRUCTOR ("")
12    end function
13
14    function CONSTRUCTOR (PlaintextCharacters)
15      try
16        lock thread
17        SHA2Hash <- SHA2 HASH (PlaintextCharacters)
18        SHA3Hash <- SHA3 HASH (PlaintextCharacters)
19        WhirlpoolHash <- WHIRLPOOL HASH (PlaintextCharacters)
20        ENCRYPT CHARACTERS (PlaintextCharacters)
21      end try
22      finally
23        lock thread
24        for all Character ∈ PlaintextCharacters do
25          Character <- 0
26        end for
27      end finally
28    end function
29
30    function ENCRYPT CHARACTERS (PlaintextCharacters)
31      try
32        lock thread
33        PlaintextBytes <- TO BYTE ARRAY (PlaintextCharacters)
34        ENCRYPT BYTES (PlaintextBytes)
35      end try
36      finally
37        lock thread
38        for all Byte ∈ PlaintextBytes do
39          Byte <- 0
40        end for
41      end finally
42    end function
43
44    function ENCRYPT BYTES (PlaintextBytes)
45      lock thread
46      GENERATE KEY (LENGTH (PlaintextBytes))
47      for i ∈ (0, LENGTH (PlaintextBytes)) do
48        Ciphertext[i] <- PlaintextBytes[i] XOR Key[KeyOffset + i]
49      end for
50    end function
51
52    function GENERATE KEY (PlaintextLength) {
53      lock thread
54      Ratio <- TRNG (2.25, 9.25)
55      KeyLength <- PlaintextLength * Ratio
56      KeyOffset <- TRNG (0, KeyLength - PlaintextLength)
57      for i ∈ (0, KeyLength) do
58        Key[i] <- TRNG ()
```

```
    end for
  end function

  function DECRYPT CHARACTERS
    try
      lock thread
      PlaintextBytes <- DECRYPT BYTES
      return TO CHARACTER ARRAY (PlaintextBytes)
    end try
    finally
      lock thread
      for all Byte ∈ PlaintextBytes do
        Byte <- 0
      end for
    end finally
  end function

  function DECRYPT BYTES
    lock thread
    for i ∈ (0, LENGTH (Ciphertext)) do
      PlaintextBytes[i] <- Ciphertext[i] xor Key[KeyOffset + i]
    end for
    return PlaintextBytes
  end function

  interface ACCESSOR
    function ACCESS(PlaintextCharacters)
  end interface

  function ACCESS(Accessor)
    try
      lock thread
      PlaintextCharacters <- DECRYPT CHARACTERS
      Accessor.ACCESS (PlaintextCharacters)
    end try
    finally
      lock thread
      for all Character ∈ PlaintextCharacters do
        Character <- 0
      end for
    end finally
  end function

  function APPEND CHARACTER (Character)
    INSERT CHARACTER AT (Character, LENGTH (Ciphertext))
  end function

  function APPEND CHARACTERS AT (Characters)
    INSERT CHARACTERS AT (Characters, LENGTH(Ciphertext))
  end function

  function INSERT CHARACTER AT (Character, Position)
    INSERT CHARACTERS AT (Character, Position)
  end function

  function INSERT CHARACTERS AT (Characters, Position)
    lock thread
    if Position ∈ (-∞,0) ∪ (LENGTH (Ciphertext),∞) then
      return
    end if
```

```
120       for i ∈ (0, KeyOffset + Position) do
121         NewKey[i] <- Key[i]
122       end for
123       for i ∈ (0, LENGTH (Characters)) do
124         NewKey[KeyOffset + Position + i] <- TRNG()
125       end for
126       for i ∈ (KeyOffset + Position, LENGTH (Key)) do
127         NewKey[i + LENGTH (CHARACTERS)] <- Key[i]
128       end for
129       for i ∈ (0, LENGTH (Key)) do
130         Key[i] <-0
131       end for
132       Key <- NewKey
133
134       for i ∈ (0, Position) do
135         NewCiphertext[i] <- Ciphertext[i]
136       end for
137       for i ∈ (0, LENGTH (Characters)) do
138         NewCiphertext[Position + i] <- Characters[i] XOR Key[KeyOffset +
139   Position + i]
140       end for
141       for i ∈ (Position, LENGTH (Ciphertext)) do
142         NewCiphertext[i + LENGTH (Characters)] <- Ciphertext[i]
143       end for
144       for i ∈ (0, LENGTH (Ciphertext)) do
145         Ciphertext[i] <- 0
146       end for
147       Ciphertext <- NewCiphertext
148
149       UPDATE ()
150     end function
151
152     function REMOVE CHARACTER AT (Position)
153       REMOVE CHARACTERS AT (Position, 1)
154     end function
155
156     function REMOVE CHARACTERS AT (Position, Length)
157       lock thread
158       if Position ∈ (-∞,0) ∪ (LENGTH (Ciphertext) - Length,∞) then
159         return
160       end if
161       if Length ∈ (-∞,0) ∪ (LENGTH (Ciphertext) - Position,∞) then
162         return
163       end if
164
165       for i ∈ (0, KeyOffset + Position) do
166         NewKey[i] <- Key[i]
167       end for
168       for i ∈ (KeyOffset + Position, LENGTH (Key) - Length) do
169         NewKey[i] <- Key[i + Length]
170       end for
171       for i ∈ (0, LENGTH (Key)) do
172         Key[i] <-0
173       end for
174       Key <- NewKey
175
176       for i ∈ (0, Position) do
177         NewCiphertext[i] <- Ciphertext[i]
178       end for
179       for i ∈ (Position, LENGTH (Ciphertext) - Length) do
```

```
      NewCiphertext[i] <- Ciphertext[i + Length]
    end for
    for i ∈ (0, LENGTH (Ciphertext)) do
      Ciphertext[i] <- 0
    end for
    Ciphertext <- NewCiphertext

    UPDATE ()
  end function

  function SET CHARACTER AT (Character, Position)
    SET CHARACTERS AT (Character, Position)
  end function

  function SET CHARACTERS AT (Characters, Position)
    lock thread
    if Position ∈ (-∞,0) ∪ (LENGTH (Ciphertext) - LENGTH
(Characters),∞) then
      return
    end if
    if LENGTH (Characters) ∈ (-∞,0) ∪ (LENGTH (Ciphertext) -
Position,∞) then
      return
    end if

    for i ∈ (0, LENGTH (Characters)) do
      Key[KeyOffset + Position + i] <- TRNG()
    end for
    for i ∈ (0, LENGTH (Characters)) do
      Ciphertext[Position + i] <- Key[KeyOffset + Position + i] XOR
Characters[i]
    end for

    UPDATE ()
  end function

  function COPY
    lock thread
    for i ∈ (0, LENGTH (Ciphertext)) do
      Ciphertext2[i] <- Ciphertext[i]
    end for
    for i ∈ (0, LENGTH (Key)) do
      Key2[i] <- Key[i]
    end for
    KeyOffset2 <- KeyOffset
    SHA2Hash2 <- SHA2Hash
    SHA3Hash2 <- SHA3Hash
    WhirlpoolHash2 <- WhirlpoolHash

    SecureString <- CONSTRUCTOR
    SecureString.Ciphertext <- Ciphertext2
    SecureString.Key <- Key2
    SecureString.KeyOffset <- KeyOffset2
    SecureString.SHA2Hash <- SHA2Hash2
    SecureString.SHA3Hash <- SHA3Hash2
    SecureString.WhirlpoolHash <- WhirlpoolHash2
    return SecureString
  end function

  function UPDATE ()
    ACCESS (ACCESSOR
```

```
241         function ACCESS (PlaintextBytes)
242           SHA2Hash <- SHA2 HASH (PlaintextCharacters)
243           SHA3Hash <- SHA3 HASH (PlaintextCharacters)
244           WhirlpoolHash <- WHIRLPOOL HASH (PlaintextCharacters)
245         end function
246       )
247     end function
248
249     function EQUALS (SecureString)
250       return VERIFY SHA2Hash (SHA2Hash)
251     end function
252
253     function VERIFY SHA2Hash (Hash)
254       return SHA2Hash = Hash
255     end function
256
257     function VERIFY SHA3Hash (Hash)
258       return SHA3Hash = Hash
259     end function
260
261     function VERIFY WhirlpoolHash (Hash)
262       return WhirlpoolHash = Hash
263     end function
264
265     function GET SHA2 HASH ()
266       return SHA2Hash
267     end function
268
269     function GET SHA3 HASH ()
270       return SHA3Hash
271     end function
272
273     function GET WHIRLPOOL HASH ()
274       return WhirlpoolHash
275     end function
276
277     function CLEAR
278       lock thread
279       for i ∈ (0, LENGTH (Key)) do
280         Key[i] <-0
281       end for
282       for i ∈ (0, LENGTH (Ciphertext)) do
283         Ciphertext[i] <- 0
284       end for
285       KeyOffset <- 0
286     end function
287
288   end class
```