FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Bezpečnostní analýza programu IrfanView
Student:	Bc. Radek Jizba
Vedoucí:	Ing. Josef Kokeš
Studijní program:	Informatika
Obor / specializace:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1) Seznamte se s prohlížečem obrázků IrfanView.
2) Technikami reverzní analýzy prozkoumejte klíčové části programu. Zaměřte se na obvyklé bezpečnostní nedostatky - přetečení bufferu, použití neinicializovaných proměnných a podobně.
3) Dále nastudujte největší bezpečnostní incidenty programu v minulosti. Proveďte základní kontrolu, jak je tvůrce programu vyřešil a zda podnikl kroky pro zabránění podobným incidentům v budoucnosti.
4) Vyhodnoťte svá zjištění, formulujte závěry o kvalitě programu z pohledu bezpečnosti.

Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 7. února 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Bezpečnostní analýza programu IrfanView

Bc. Radek Jizba

Katedra informační bezpečnosti
Vedoucí práce: Ing. Josef Kokeš

2. května 2022

Poděkování

Tímto bych chtěl poděkovat Ing. Josefu Kokešovi za odborné vedení své diplomové práce, poskytování cenných připomínek a námětů na zlepšení textu a za čas věnovaný konzultacím.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel licenční smlouvu o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 2. května 2022 . . . . . .

Odkaz na tuto práci

Jizba, Radek. *Bezpečnostní analýza programu IrfanView*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato diplomová práce se zabývá analýzou programu na prohlížení obrázků IrfanView. Taktéž jsou v této práci zhodnoceny opravy předchozích zranitelností tohoto programu. Výsledkem této práce je zjištění, že je program IrfanView velmi dobře udržován a autor v krátkém čase opravuje nalezené zranitelnosti.

Klíčová slova IrfanView, analýza programu, CVE, CVSSv2, IDA Free, x64dbg

Abstract

This Thesis focuses on analysis of image viewing software IrfanView. Previous vulnerabilities and their fixes of this software were also mapped and analyzed. As a result this thesis comes to a conclusion, that software IrfanView is very well maintained by the author, and all vulnerabilities are fixed in very little time.

Keywords IrfanView, program analysis, CVE, CVSSv2, IDA Free, x64dbg

Obsah

Seznam obrázků

Úvod

V aktuální době je velmi populární vývoj programů pro různé platformy a příležitosti. To ovšem způsobuje, že značné množství špatně napsaných nebo přímo nebezpečných programů je používáno běžnými uživateli. Přestože to vývojář nemusel tak zamýšlet, mohou se tyto programy v rukou útočníka stát další potencionálně nebezpečnou zbraní. Nejedná se ovšem pouze o nově napsané programy. Tato oblast informačních technologií se neustále vyvíjí a je tedy možné, že program, který byl v minulosti považován za dokonale napsaný, obsahuje zranitelnost, o které ještě nikdo neví. Z tohoto důvodu je nutné analyzovat všechny používané programy, aby se počet těchto zranitelností co nejvíce zmenšil.

IrfanView je software na prohlížení a drobnou editaci různých druhů souborů, primárně obrázků. Hlavním cílem vývojáře tohoto softwaru je vytvoření snadno použitelného programu pro začátečníka a velmi silného nástroje pro pokročilé uživatele. Dále lze říci, že tento software se snaží být stále aktuální, o čemž svědčí jeho neustálý vývoj. Navíc se tento autor chlubí prvenstvím v zobrazování souborů GIF na OS Windows a je jedním z prvních softwarů na zobrazení vícestránkového souboru typu TIF [1]. V této práci bude analyzována verze 4.58 programu IrfanView.

Cílem práce je seznámení s programem IrfanView, který se využívá primárně k prohlížení obrázků či jiných souborů s grafickou reprezentací. Jedná se o program se zaměřením jak na pokročilé uživatele, tak na začátečníky podporující různé druhy pluginů. Vlastní software je z pohledu moderního návrhu softwaru poměrně starý, což může způsobovat kombinování starého a nového zdrojového kódu. Takto vytvořený kód je velmi častým zdrojem chyb, a proto značná část této práce bude zaměřená na analýzu vlastního programu. Jak bylo psáno výše, program je velmi modulární a pro čtení některých formátů spoléhá na různé pluginy ostatních autorů. Některé z těchto pluginů jsou instalovány automaticky, jiné vyžadují dobrovolný souhlas s instalací rozšíření. Protože program je vytvořen více autory, lze předpokládat, že bude obsaho-

vat více chyb než jiné. Posledním cílem této diplomové práce je analyzovat předchozí zveřejněné chyby a zjistit, zdali je autor již opravil, případně jak je opravil.

Kapitola 1

Analýza

Při analýze softwaru je nutné zvolit správný přístup k zadané problematice. Pokud se bude jednat o škodlivý druh softwaru, bude pravděpodobně velice nežádoucí tento software spouštět. Naopak pokud si můžeme být jistí, že tvůrce softwaru se nesnaží vědomě poškodit uživatele jeho softwaru, či od něj získat nějaké informace, je výhodné tento software spustit. V běžícím programu je mnohem snadnější zjistit, co určité části dělají, neboť je možné toto chování přímo pozorovat. Těmto dvěma přístupům se říká statická a dynamická analýza programu. Analýza ovšem není nikdy takto černobílá, a tak bude definován ještě jeden přístup, který se nazývá hybridní. Kupříkladu při analýze softwaru, který při svém běhu komunikuje s nějakou jinou entitou, který získává klíče k rozšifrování některých částí, není možné stoprocentně analyzovat program statickou analýzou. Hybridní analýza se tedy snaží spojit tyto dva přístupy a získat výhody obou dvou.

1.1 Dynamická analýza

Jak již název napovídá, při této analýze je třeba zkoumat spuštěný program. Je tedy třeba nějak informovat operační systém o úmyslu editace a zkoumání programu s využitím jiného programu. K tomu se používá debugovací aplikační programové rozhraní, které vytvoří závislost typu rodič-dítě mezi dvěma procesy. Tohoto lze docílit dvěma základními způsoby. Připojením debugeru za běhu, nebo spuštěním programu přímo přes debugger. Obě možnosti potřebují takzvanný debugovací příznak (debug flag), který umožní výše zmíněné operace. Nyní je již možné nakládat se zkoumaným programem dle libosti. Je ovšem třeba dávat pozor na antidebugging techniky, které občas někteří tvůrci programů využívají [2]. Pro dynamickou analýzu je třeba využívat některých nástrojů, které debugging poskytuje. Mezi tyto nástroje patří softwarové nebo hardwarové breakpointy.

Softwarový breakpoint Softwarový breakpoint je instrukce vyvolávající výjimku při jejím spuštění. Vyvolání jedné z těchto výjimek je možné docílit pomocí instrukce INT 3, kterou využívají debuggery pro zastavení provádění strojového kódu. Pokud se tedy vyskytne výjimka tohoto druhu, je možné zjistit, v jakém stavu se aktuálně program nachází, a případně jej upravovat. Při postupném krokování se používá tato instrukce také a tím je docíleno, že program spustí pouze jednu instrukci a opět se zastaví. Toto chování je ovšem možné detekovat kupříkladu pomocí měření času výpočtu nějaké části kódu, jak je popsáno v sekci 1.4.5.

Takovýchto breakpointů může být v programu teoreticky nekonečně mnoho, neboť se v realitě jedná pouze o rozšíření původního programu o instrukci. Není tedy třeba aktivně tyto breakpointy sledovat, neboť procesor sám upozorní debugger, když na takovouto instrukci narazí.

Naopak hlavní nevýhodou tohoto druhu breakpointu je nutnost spuštění instrukce INT 3. Pokud je tedy třeba sledovat nějakou adresu v paměti, protože je složité zjistit, kdy se tato adresa upravuje, tento druh breakpointu není vhodný.

Hardwarový breakpoint Hardwarový breakpoint je možné použít k dosažení podobného výsledku jako softwarový, ale to není jeho primární funkcí. Tyto breakpointy mohou být použity na jakékoliv nevolatilní části paměťového subsytému a mohou monitorovat jakoukoliv manipulaci s tímto místem. Těchto breakpointů je ovšem omezené množství, neboť vyžadují podporu na úrovni procesoru. Procesor zde tedy monitoruje jakýkoliv přístup k zadaným adresám a v případě přístupu vytvoří výjimku pro debugger.

Hlavní výhodou hardwarového breakpointu je možnost monitorování téměř jakékoliv části paměti s ohledem na přístup, čtení nebo zápis. Pokud tedy je velmi obtížné zjistit, kdy program pracuje s nějakou částí paměti, pomocí hardwarového breakpointu je tato informace velmi snadno zjistitelná.

Stejně jako softwarový breakpoint je možné odhalit hardwarové breakpointy díky měření času výpočtu. Další nevýhodou je omezený počet těchto breakpointů kvůli omezení architektury.

1.2 Statická analýza

Statická analýza je velmi přímočará. Zde je možné prohlížet spustitelný soubor ve tvaru, v jakém je zapsán na disku v počítači. Bohužel při tomto přístupu není možné snadno řešit různá volání funkcí, která závisejí na nějakých faktorech. Kupříkladu je velice nepraktické analyzovat program používající třídy a členské funkce, protože ve skutečnosti bude při statickém pohledu na spustitelný soubor pouze instrukce call na registr. Většinou je samozřejmě možné toto volání manuálně získat, ale nejedná se o snadnou práci. Čistě statickou analýzu navíc nelze použít u různých zabalených (packed) binárních souborů.

Tyto soubory většinou získávají klíče z externího zdroje a teprve při spuštění se rozbalí do použitelné formy [3].

Nejdůležitější částí statické analýzy je nutnost redukce disassemblovaného kódu. Přímá analýza bez jakýchkoliv předchozích příprav je velmi neefektivní a může vést ke zbytečné práci. Z tohoto důvodu je vhodné nejprve identifikovat kupříkladu importované funkce, zjistit, jaké řetězce jsou v programu používané, a ověřit si, z jakých částí se program skládá. Po identifikaci těchto míst je teprve vhodně začít s analýzou.

1.3 Použité nástroje

Pro analýzu programu není třeba využívat téměř žádného externího programu. Ve skutečnosti jediné, co je zapotřebí, je nějaký prohlížeč souborů, který může interpretovat data jako hexadecimální číslice. V dnešní době ovšem existuje mnoho moderních nástrojů, díky kterým je velká část analýzy velmi automatizovaná. Kupříkladu disassemblery dokáží vlastní hexačísla přímo převádět na instrukce, které jsou lépe čitelné pro potřeby analýzy. Samozřejmě je důležité si uvědomit, že přechodem mezi těmito zobrazeními se mohou nějaké informace ztratit nebo minimálně nemusí být tak jasně čitelné. Toto chování využívají takzvané obfuskace, které dovolují autorům software co nejvíce ztížit vlastní analýzu.

1.3.1 IDA

IDA je nástroj sloužící primárně na statickou analýzu zdrojového kódu. Dovoluje uživateli zobrazit všechny důležité informace o spustitelném souboru. Mezi tyto informace patří seznam řetězců, seznam funkcí nebo například importy z ostatních knihoven. Největší nedostatek programu IDA je nemožnost zobrazení takzvané packované binárky. Pokud je potřeba analyzovat nějakou takto dynamicky se měnící binárku, je třeba využít nějakých externích nástrojů. Vlastní program IDA má nově zabudovaný i debuger, který by nám mohl s tímto rozbalením pomoci, ale ten zatím není tak dobrý jako některé jiné dedikované nástroje. V práci je použita pouze volně dostupná IDA. Oproti IDA Pro zde nelze scriptovat nebo analyzovat určité formáty (jako jsou například programy pro ARM architekturu).

1.3.2 x64dbg

Tento program je poměrně pokročilý debugger s podporou běžných pluginů. Tento software má velice mnoho nástrojů na analýzu programu. Umožňuje stejně jako IDA zobrazení disassemblovaného kódu ve tvaru grafů, díky kterému je velice snadné analyzovat jednotlivé části souboru. Dále umožňuje zobrazení seznamu exportů a importů, což je velmi vhodné pro dynamickou analýzu. Programy používající různé pluginy, které se načítají až při běhu programu,

tedy zobrazí pomocí x64dbg všechny takto načtené knihovny. Velmi podstatnou částí tohoto programu je, že dovoluje zobrazení stringů a podobných zajímavých sekvencí. V neposlední řadě je vhodné zmínit nástroj Scyla, který dovoluje vytvořit novou verzi spuštěného souboru. Tento nástroj se primárně používá při analýze takzvaných packovaných programů. Většinou jsou packované programy nějaký druh malware s vysokou entropií.

1.3.3 CFF Explorer

Jak bylo zmíněno výše, spustitelné soubory obsahují značné množství informací. K snadné extrakci těchto informací slouží právě nástroj CFF Explorer. Velice snadno je díky tomuto programu vidět, kdo je autorem daného binárního souboru, jaký kompilátor byl použit nebo například s jakým offsetem jsou mapované funkce v paměti.

1.3.4 Resource Hacker

Resource Hacker slouží primárně k extrakci různých zdrojů z binárních souborů. Velké množství programů má v sobě ukryté vzory, kupříkladu pro dialogová okna. Není tedy třeba zdlouhavě vždy znovu definovat dimenze, místo a lokaci takového objektu. Ve zdrojovém kódu stačí použít funkci Load Resource. Resource Hacker pouze proskenuje spustitelný soubor a nalezne všechny tyto zdroje.

1.3.5 Process Hacker

Je program, který dovolí hlubší náhled do běžícího programu než Task manager nebo Performance monitor. Dovolí totiž kupříkladu sledovat počet přístupů do paměti, nastavit těmto operacím priority a podobně.

1.3.6 HxD

HxD je jednoduchý nástroj na editaci souborů. Jeho použití při této analýze je velmi kritické, neboť je třeba editovat binární data souborů. Tento nástroj je samozřejmě nahraditelný jakýmkoli jiným hexa editorem, ovšem HxD byl zvolen pro jeho jednoduchou manipulaci s editovanými daty.

1.3.7 OllyDbg

OllyDbg je jednoduchý debugger, který je vhodný na analýzu malých projektů. Zároveň má jiné algoritmy na rozpoznávání některých konstuktů, a je tedy možné říci, že se jedná v některých speciálních případech o lepší program na debugging nežli výše zmíněný program x64dbg.

1.4 Obrana proti debuggingu a obfuskace

Je mnoho způsobů, jakými se tvůrci software brání proti reverse engineerovaní těchto programů. Primárně takovéto chování je vidět u různých druhů malware, ale je možné se s těmito praktikami setkat i u legitimního software. Jedním takovým příkladem jsou různé drivery nebo firmware. Neslavně je těmito praktikami proslulá firma NVIDIA a jejich linuxové drivery [4].

1.4.1 Packování

Pokud chce tvůrce programu co nejvíce zamaskovat vlastní obsah programu (API volání strukturu apod), použije techniku nazvanou packování nebo šifrování. Jedná se o techniku, prostřednictvím které program zamaskuje volání funkcí díky šifrování. Takto vytvořený program má nezašifrovaný začátek, aby jej bylo možné spustit. Tato úvodní část následně rozbalí zbytek programu, aby bylo možné ho spustit. Detekce takto obfuskovaného software je poměrně snadná. Většina takovýchto programů má totiž poměrně vysokou entropii a je možné je tedy snadno rozpoznat. Toto ovšem neříká nic o obsahu vlastního zdrojového souboru. Navíc tvůrci malware velmi často využívají těchto vlastností, a když je jejich malware označen jako škodlivý nějakou antivirovou společností, může velice snadno přegenerovat zašifrovanou část, a to je většinou dostatečná změna, aby různé detekční programy považovaly tento program za úplně nový [3].

1.4.2 Maskování

Ve strojovém kódu se nacházejí různé konstrukty, které je možné využít k maskování některých instrukcí. Jedním takovým příkladem je instrukce skoku jmp, která skáče sama do sebe. Toto chování je dostatečné pro zmatení velkého množství syntaktických analyzátorů, které se většinou pokusí rozklíčovat takovéto volání a špatně určí volání funkcí a instrukce. Toto je ovšem při lineární analýze problematické. Tento druh obfuskace je ovšem relativně snadné obejít jednoduchými scripty na úpravu zdrojového souboru [5].

1.4.3 Neprůhledné predikáty

Další možnost, jak ztížit analýzu, je vytvoření takzvaných neprůhledných predikátů. Jedná se o využití predikátů, které nejsou pro analytika příliš průhledné, ale tvůrce software dopředu ví, jak vyhodnocení predikátu dopadne. Jedná se kupříkladu o predikát, který je vždy pravdivý. Při analýze takového programu je většinou velice složité prohlédnout takto vytvořený predikát, a tak je občas nutné prohledat slepou větev. Tím se ovšem ztěžuje analýza daného kusu programu. Navíc, pokud je tato obfuskace napsaná dobře, je téměř nemožné tento predikát prohlédnout, pokud nebudme simulovat běh tohoto programu. Toto vychází z klasikého Halting problému [5].

1.4.4 Detekování debug příznaku

Každý debugovaný program má takzvaný debug flag. Jednou z antidebugovacích technik je detekce tohoto flagu. Pokud je tento flag detekován, může program skončit. Takovéto chování je ovšem z hlediska analýzy docela průhledné a je tedy spíše vhodné použít tuto informaci jako vstup do slepé větve programu. Pokud si totiž při analýze nevšimneme takového konstruktu, je velmi snadné protáhnout dobu analýzy programu [2].

1.4.5 Časování

Jedním z dalších způsobů, jak detekovat přítomnost cizího programu, je vytvořit nějakou nicnedělající smyčku. Na začátku této smyčky si stačí uložit procesorový čas a vypočítat, jak dlouho trvalo zpracování této smyčky. Při debugingu je vlastní výpočet trochu opožděn a je tedy možné detekovat přítomnost debugovacího programu. Nejtěžší z hlediska autora software je vytvořit správný časový rozsah takový, aby většina uživatelů tímto „benchmarkem“ prošla, ale aby co nejvíce snížila možnost průchodu s použitým debugerem [2].

1.4.6 Eliminace volální knihovních funkcí

Jednou z hlavních zbraní při analýze programu je pohled na importované funkce programu. Z nich je totiž většinou velice snadné zjistit, jaké rámcové chování lze od programu očekávat. Proto je pro tvůrce takového obfuskovaného programu velmi důležité tyto informace co nejvíce eliminovat [6]. Tohoto chování docílí vývojář převážně vytvořením vlastních alternativních funkcí k již stávající knihovní funkci.

1.4.7 Rozbalování

Rozbalování je jednou z technik, na kterou se nelze dívat jako na techniku ztěžující analýzu úmyslně. Většinou je totiž při návrhu software vhodné optimalizovat vše pro cache a podobné konstrukce. Toto ovšem může velmi zvětšit výslednou velikost vlastního programu a občas tím pádem ztížit vlastní analýzu. Na druhou stranu různé metody využívající různých moderních instrukcí procesoru, jako je třeba looptiling nebo loop unrolling, jsou velmi výhodné z hlediska doby běhu programu.

Kapitola 2

Strojový kód

Assembler neboli jazyk symbolických adres je přepis strojového kódu do snadněji čitelného jazyka. V této kapitole bude strojový kód rozebrán jako kompilovaný kód vyššího jazyka, jmenovitě C++, MSVC++ (Dekorace). Velmi důležité je uvědomit si, že při práci s tímto jazykem je třeba operovat s omezeným počtem registrů a omezeným počtem instrukcí. Na rozdíl od kompilace je přepis ze strojového jazyka do Assembleru bezztrátový. To znamená, že se neztratí žádná informace, a pokud bude uživatel chtít, může z Assembler znovu vygenerovat strojový kód, který bude stejný jako původní. Jednou z nevýhod je možnost obfuskace, která může ztížit práci analytika. Více o této problematice v kapitole Obrana proti debuggingu a obfuskace. Vzhledem k úzkému svázání Assembleru se strojovým kódem není možné předpokládat, že Assembler vypadá vždy stejně. Díky vývoji moderních komponent (v tomto případě si lze představit procesor) se snaží vývojáři přidat nové instrukce, které zvýší výkonnost vlastního procesoru. To ovšem znamená, že ne všechny procesory mají stejnou instrukční sadu, takzvanou ISA (Instruction set architecture). Assembler lze tedy dělit dle různého hardware, pro který je napsaný. Nejjednodušší rozdělení je na RISC (redukovanou instrukční sadu) a CISC (komplexní instrukční sadu). Redukovaná instrukční sada má velmi omezený počet instrukcí, protože je jasně specifikováno, jak dlouhé musí instrukce být. Dále je RISC více závislý na použití kvalitní paměti oproti CISC. Jeho primární výhoda ovšem spočívá v menší spotřebě energie, a proto se používá primárně v serverových zařízeních nebo IOT. Oproti tomu CISC je velmi komplexní instrukční sadou a obsahuje velké množství instrukcí pro velmi specifické použití. Za vyšší verzatilitu se ovšem platí větší spotřebou energie (větším ohříváním komponent) [7]. Tato sada nemá přesně danou velikost instrukční sady, a tak je díky tomu velmi snadné (pro vývojáře hardware) ji rozšiřovat. Nejrozšířenější komplexní instrukční sadou je jednoznačně x86/x64. Zde je důležité zmínit, že x86 je Assembler napsaný primárně pro 16bitovou architekturu. Tato zařízení jsou dnes již spíše v úpadku na trhu s osobními počítači. Oproti tomu x64 je psán pro 64bitovou architekturu.

64bitová architektura je navíc zpětně kompatibilní s programy pro 32bitovou architekturu. Vlastní x64 architektura, potažmo Assembler, je stále vyvíjena a jsou do ní přidávány různé nové instrukce. Mezi tyto nové instrukce patří kupříkladu takzvané vektorové instrukce, které dokáží pracovat s více bloky paměti zároveň. Jedná se kupříkladu o vektorovou sadu AVX-256, AVX-512, MMX nebo SSE. Nejedná se o pravidlo, ale je možné předpokládat, že program napsaný pro 32bitovou architekturu nemusí využívat optimálních struktur pro 64bitové procesory. Proto je občas vidět absence těchto pokročilých instrukcí u programů primárně napsaných pro nějakou předchozí architekturu. Nejedná se o závažný bezpečnostní problém, ale díky těmto drobným nedostatkům se může velmi zpomalit výpočet určitých druhů programů, primárně takových, které dělají rozsáhlé výpočty nad velkými maticemi, jako jsou například různé editory obrázků nebo videa [8].

Assembler nemá jednotnou formu zápisu. Ve skutečnosti existují dva hlavní přístupy, jak zapisovat instrukce tohoto jazyka. Jedná se o notaci od firmy Intel a o notaci od firmy AT&T. Obě notace jsou ekvivalentní s ohledem na sdělovací schopnosti a jedná se tedy spíše o osobní preferenci. U notace od firmy Intel je vždy cílový argument první. Kdežto u AT&T notace je cílový argument poslední. Není příliš složité rozlišit, která z notací je v aktuálně prozkoumávaném kusu kódu použita, neboť instrukce mohou vypadat trochu jinak, před registry se u AT&T píší procenta, a před čísly se v AT&T píše dolar. V této práci budou veškeré Assembler kódy zapsané v Intel notaci.

2.1 Registry

Procesorové registry jsou jednou z mnoha předsunutých pamětí v paměťovém modelu zařízení. Jedná se o nejrychlejší část paměti s velmi omezenou kapacitou. Ve skutečnosti existuje mnoho různých druhů registrů, jako jsou například GPR, FPR, nebo vektorové registry. V této kapitole nebudou představeny všechny registry, ale pouze ty nejvíce využívané při analýze programu. První pohled bude na vlastnosti registrů na 32bitovém systému a jejich chování a následně bude vše rozšířeno na 64bitovou verzi.

Neprve je důležité rozdělit registry na registry v x86 architekturách a na rozšíření registrové sady, které přišlo s x64 architekturou. Základní registry pro x86 architekturu jsou **EAX**, **EBX**, **ECX**, **EDX**, **EBP**, **EDI**, **ESP** a **EIP**. Registr **EAX** je jeden z nejdůležitějších registrů při analýze zdrojového kódu, neboť většinou obsahuje návratovou hodnotu funkce. Pokud funkce žádnou návratovou hodnotu nemá specifikovanou (vrací typ void), není obsah registru **EAX** definován. Registry **EBX**, **ECX** a **EDX** nemají ve většině případů žádný speciální význam a používají se jako pomocné registry. **ESP** (stack pointer) je registr ukazující na vrchol zásobníku (stacku). Pomocí tohoto registru se přistupuje k lokálním proměnným a adresám, které aktuálně nejsou na haldě nebo nahrané v nějaké jiném registu. Registr **EBP** (base pointer)

ukazuje na začátek stacku patřícího aktuálnímu tělu funkce. Ve skutečnosti se jedná o předchozí hodnotu **EBP**. Na nižších adresách se poté vyskytují lokální proměnné, kanárek, návratová adresa a na vyšších argumenty. **ESI** je registr, který se používá hlavně při indexování. Toto ovšem není pravidlo a není tedy možné na toto chování spoléhat. Nakonec registr **EIP** (instruction pointer) ukazuje na aktuálně prováděnou část kódu [9].

Obrázek 2.1: Typická struktura zásobníku programu.

Tyto registry se vyskytují i u 64bitové architektury, kde mají podobné vlastnosti jako na 32bitové. 64bitové architektury také zvětšují počet těchto „základních“ registrů o 8. Tyto registry již nemají speciální pojmenování jako

registry převzané z 32bitových architektur a označují se jendoduše **R8** – **R15**.

Vzhledem k urychlení výpočtu není třeba všechny argumenty nahrávat na stack. Při nahrávání na stack se vytváří nemalá režie při zápisu do „hlavní paměti“, a tak se první 4 argumenty nahrávají pouze do registrů. V operačním systému Windows pro toto chování byly zvoleny registry **RCX**, **RDX**, **R8** a **R9**. Pokud má volaná funkce více než 4 argumenty, nezbývá nic jiného než zbylé argumenty nahrát na zásobník. Když je ale argumentů méně než 4, zůstává obsah těchto registrů nepozměněn (nedefinovaná hodnota). V neposlední řadě je nutné zmínit, že při volání členské fukce nějaké třídy, ve které je možné použít ukazatel this, je vždy první argument (**RCX**) obsazen tímto ukazatelem. Tedy, i když takováto členská funkce nebude mít jediný argument, stále se při zavolání na místo prvního argumentů nahraje tento ukazatel.

Co se týká návratových hodnot a registrů, mají předchozí odstavce dostatek informací k pochopení velké části programu. Existuje ovšem mnoho dalších registrů, které je možné použít. Kupříkladu již dříve zmiňované registry na vektorové operace ve formě **XMM0**, **YMM0** nebo například **ZMM0**. Jako ukázku jsem zvolil právě tyto registry, protože je u nich možné stejně jako u **EAX** uložit návratovou hodnotu. Zde ovšem je samozřejmě nutné přesně vědět, o jakou verzi procesoru se jedná, neboť ne všechny procesory mají kupříkladu rozšíření AVX-512.

V předchozích odstavcích bylo referováno o registrech jako o proměnné, jejíž označení většinou začíná písmenem **E**. Ve skutečnosti, pokud je potřeba využít celé šířky registru, tak se na 64bitové architektuře **E** nahradí za **R**. Tímto je možné přistoupit ke všem 64 bitům daného registru. Pokud je použito pouze **E**, lze adresovat spodních 32 bitů tohoto registru. Pokud je předpona vynechána, adresuje se pouze spodních 16 bitů. A v neposlední řadě, pokud je použita předpona **H**, je adresováno předposledních 8 bitů a pokud **L**, je adresováno posledních 8 bitů. Stejně tak lze adresovat nižší bity vektorových registrů pomocí podobných pravidel [9].

2.2 Instrukce

Assembler je dle standardních měřítek velmi nízký jazyk, neboli má velmi malou úroveň abstrakce. K tomuto závěru je možné dojít velmi jednoduše, neboť se ve skutečnosti jedná pouze o přepis strojového kódu do formátu lépe pochopitelného pro člověka. I tak je ovšem důležité znát některé instrukce a jejich pravé hodnoty při analýze těchto programů.

Skokové instrukce Jednou z nejdůležitějších skupin instrukcí jsou instrukce měnící tok kódu. Jinými slovy instrukce měnící hodnotu v registru **RIP** neboli Instruction pointer. Pro tuto práci není příliš důležité rozebírat, jak tyto instrukce fungují na úrovni procesoru, stačí si pouze uvědomit, jaký mají výsledek při provedení.

Skoky lze rozdělit na dvě hlavní skupiny, a sice podmíněné a nepodmíněné skokové instrukce. Nepodmíněný skok je, jak již název napovídá, taková instrukce, která vždy při provedení změní hodnotu **RIP** na nějakou jinou. Oproti tomu podmíněné skokové instrukce jsou závislé na nějaké další informaci, jmenovitě se jedná o takzvané příznaky neboli flagy. Jedná se o příznaky **CF** (Carry flag), **PF** (Parity flag), **ZF** (Zero flag), **SF** (Sign flag) a **OF** (Overflow flag). Toto nejsou samozřejmě všechny příznaky, které většina procesorů nabízí, ale pro tuto práci jsou dostatečné.

Podmíněné skokové instrukce se vyskytují ve dvou hlavních formách, pro krátké a dlouhé skoky. Toto je ovšem trochu zavádějící rozlišení. Krátké skokové intrukce mění **RIP** na adresu relativní vůči aktuální **RIP**. Tyto skoky jsou ovšem omezené vzdáleností, protože mohou měnit **RIP** pouze v rámci aktuálního segmentu. V dnešním kódu je toto chování spíše vzácné. Oproti tomu takzvané dlouhé skokové instrukce mění **RIP** na jakoukoliv jinou hodnoru. To ovšem v x86 Assembler způsobí, že musí být dodána i informace o segmentu paměti, do kterého tato instrukce směřuje.

Existuje velké množství podmíněných skokových instrukcí, které je vhodné znát při analýze strojového kódu programu. V této časti práce budou představeny pouze některé z těchto instrukcí. Jednou z nejpoužívanějších skokových instrukcí je instrukce JE nebo také JZ. Tato skoková instrukce se provede, pokud jsou 2 testované argumenty rovné nebo je jeden argument nulový. Jedná se o zcela ekvivalentní zápis, neboť obě tyto skutečnosti se označují příznakem ZF nastaveným na hodnotu 1. Pro krátký skok se u této instrukce používá hodnota operačního kódu 0x74 a pro dlouhý 0x0F84. Opakem těchto instrukcí jsou instrukce JNE a JNZ (Jump not equal a Jump not zero), které se naopak provedou, pokud je řídící příznak **ZF** nastaven na 0. Hodnota OP kódu je také velmi podobná 0x75 pro krátý skok a 0x0F85 pro dlouhý. Další důležitou skokovou instrukcí je JO, která se provede, pokud přetekla proměnná. Logicky tedy kontroluje, zda příznak OF je nastaven na 1. Tato instrukce má OP kód 0x70 pro krátký skok a 0x0F80 pro dlouhý. Alternativou stejně jako u předchozího páru instrukcí je instrukce JNO s OP kódy 0x71 pro krátký skok a 0x0F81 pro dlouhý. Tyto dvě skokové instrukce nejsou příliš často viditelné, a to ani v místech, kde v minulosti byla detekována chyba způsobená přetečením. Je tedy pravděpodobné, že kompilátor tyto instrukce nahrazuje nějakou alternativou, nebo vývojář napsal program ve stylu, který nedovoluje kompilátoru snadno rozeznat, o jakou skokovou instrukci by se mělo jednat. Další důležitou skupinou podmíněných skokových instrukcí jsou instrukce reagující na znaménko proměnné. Jedná se o instrukcí JS, která se provede, pokud je testovaná proměnná záporná (neboli první bit je nastaven na 1). Tato instrukce testuje, zdali **SF** příznak je nastaven na 1 a má OP kód 0x78 pro krátké skoky a 0x0F88 pro dlouhé skoky. Alternativně lze použít opak této instrukce, instrukci JNS, která se provede, pokud je příznak **SF** nastaven na 0 a má OP kódy 0x79 pro krátký skok a 0x0F89 pro dlouhý skok. Poslední důležitou skupinou podmíněných skokových instrukcí

jsou instrukce řešící nerovnostní vazby, jako je menší a větší. Tyto instrukce je možné vidět často u částí programu, které řeší realokaci paměti nebo testují, zdali je hodnota v rámci vymezených hranic. Mezi tyto instrukce patří mimo jiné JA neboli JNBE, které se provedou, pokud je jeden z argumentů větší nežli druhý. V realitě tyto instrukce testují, zdali je příznak **CF** a **ZF** nastaven na 0. OP kód je pro tuto instukci 0x77 pro krátkou variantu skoku a 0x0F87 pro dlouhou variantu skoku. Dále je důležité zmínit, že instrukce JA (případně JNBE) porovnává neznaménkové datové typy. Těchto instrukcí je hodně a nebudou tady všechny detailně popsány, ale všechny mají velmi srozumitelné názvy a je možné tedy odvodit, kdy se tyto instrukce provedou.

K nastavení příznaků z předchozího odstavce se používá několik různých instrukcí, ale nejčastěji používaná instrukce je instrukce TEST. Tato instrukce nastavuje 3 základní příznaky **SF**, **ZF** a **PF**. **OF** a **CF** příznaky jsou vynulovány a **AF** nemá definované chování [10].

2.3 Dekorace

Při překladu zdrojového kódu do strojového je využito několik různých technik. Jednou takovou technikou je takzvaná dekorace jmen (Name decoration), která při pohledu na strojový kód může obsahovat informaci o tvaru, počtu argumentů a typu volané funkce. Většina nástrojů použitých v této práci automaticky rozpoznává tyto dekorace a dokáže tedy určit různé informace o volaných funkcích. Je důležité poznamenat, že vlastní dekorace není jednotně definovaná a každý kompilátor může mít různý formát/tvar dekorovaných jmen funkcí. Primárně je ovšem vhodné se zaměřit na kompilátor MSVC++ 8.0, neboť je to kompilátor použitý při kompilaci hlavní části zkoumaného programu analyzovaného v této práci. Typ kompilátoru při kompilaci je snadné získat přímo z binární reprezentace spustitelného souboru nebo pomocí programu CFF Explorer.

2.4 Volání funkcí

Volání funkce a návrat z této funkce je ve skutečnosti velmi triviální a je možné dosáhnout stejného výsledku pomocí instrukcí MOV a JMP. Volání funkce není v realitě nic jiného než zjištění aktuální instrukce, kterou program vykonává z registru **RIP**. Poté je následující instrukce uložena na zásobník (Stack) programu jako takzvaná návratová adresa. Nakonec je použita nepodmíněná instrukce skoku s adresou volané funkce.

Funkce končí, pokud se procesor pokusí provést instrukci RET. Instrukce RET funguje velmi podobně jako fungovala instrukce CALL. Nejdříve se získá návratová hodnota ze zásobníku, která se následně nahraje do registru **RIP**. Tímto se efektivně přesune výpočet zpět za místo volání funkce.

2.5 Volací konvence

Exituje několik volacích konvencí, které určují, jak se bude program chovat při volání nějaké funkce. V této kapitole bude rozebráno několik konvencí, které jsou velmi časté při programování v jazyce C++, a to jak u 32bitových, tak 64bitových programů. Ve skutečnosti pro 64bitové programy existuje pouze jedna volací konvence. Pro 32bitové programy je konvencí hned několik. V této kapitole budou nastíněny pouze některé z nich. Jmenovitě se bude jednat o __stdcall, __cdecl, __fastcall, __thiscall [11].

__stdcall Tento druh volání je primárně používán u Win32 aplikací, kde zásobník čistí volané funkce. Všechny argunemty jsou na stack přidávány zprava. Volaná funkce navíc maže argunety funkce ze zásobníku. Dekorace takovýchto funkcí většinou začíná podtržítkem, následuje název funkce, zavináč a velikost všech argumentů v bytech jako dekadické číslo [11]. Kupříkladu funkce

int __stdcall foo(double bar, int buz);
by vypadala takto: _foo@12

__cdecl Tato konvence je primární volací konvencí pro C a C++ programy. Narozdíl od __stdcall u této volací konvence maže argumenty volající, což může způsobit tvorbu menších programů. Argumenty jsou zapisované stejně jako u stdcall zprava doleva a stejně jako u stdcall dekorace začíná podtržítkem. Toto podtržítko není použito, pokud je daná funkce určena pro export [11].

__fastcall Fastcall je velmi podobný 64bitové konvenci volání. První dva argumenty jsou předávány pomocí registrů **ECX** a **EDX**. Dále se tato konvence chová velmi podobně jako stdcall v tom smyslu, že argumenty maže volaná funkce a při dekoraci jména funkce se používá počet bytů označující součet velikostí argumentů funkce [11].

__thiscall Thiscall je speciální volací konvence používaná u C++ programů. Chová se podobně jako fukce volaná s konvencí cdecl, protože volaný maže argumenty a čistí zásobník. Tuto konvenci je možné použít pouze, pokud volaná funkce je členskou proměnnou nějaké třídy. V registru **ECX** se předává ukazatel na instanci. Jedná se tedy o určitý kompromis mezi fastcall a cdecl [11].

64bitová konvence Volací konvence u 64bitových programů je sice pouze jedna, ale ve skutečnosti je mnohem složitější. Jak bylo psáno dříve, první 4 argumenty jsou předány v předem definovaných registrech. Primárně se tato konvence chová jako fastcall u 32bitových programů, jen používá o dva registry více. Všechny argumenty větší než 64 bitů musejí být předány refe-

rencí, není možné rozdělit jeden argument do více registrů. Navíc, pokud je předáván argument s plovoucí desetinnou čárkou, jsou použity speciální registry **XMM0L**, **XMM1L**, **XMM2L** a **XMM3L**. Je starostí volajícího vytvořit argumenty na stacku, které musejí být vždy zarovnané. Pokud nebyla provedena změna v zarovnání, je možné předpokládat, že zarovnání argumentů na zásobníku je 16 bytů. Struktury, třídy a uniony jsou předány tak, jak by byla předána celá čísla, pokud jsou ve velikosti 8, 16, 32, nebo 64 bitů. Pokud mají nějakou jinou velikost, je předán ukazatel na paměť, kde se tyto struktury nacházejí [11].

2.6 Zásobník

Zásobník je speciální část paměti programu, ve které se ukládají lokální proměnné, argumenty a podobné. Je důležité si uvědomit, že zásobník vždy roste od vyšších adres paměti směrem k nižším adresám. Struktura stacku při volání nějaké funkce v programovacím jazyce C je taková, že nejdříve se na stack přidají argumenty funkce, se kterými je funkce volána. Následně se nahraje adresa následující instrukce po zavolání této funkce. Této adrese se říká návratová adresa a při instrukce ret se na ni program vrátí. Dále se uloží záloha **EBP** a registr **EBP** začne ukazovat na tuto adresu. V neposlední řadě se provede záloha registrů, pokud je vzhledem k povaze vlastní volané funkce nutná. Nakonec se naalokují lokální proměnné dané funkce.

Tyto konstrukty tvoří základ volání funkce a její takzvané hlavičky. Z pohledu bezpečnosti byl ještě přidán koncept kanárka (nebo také sušenky), který umožní detekovat neoprávněný přepis hodnot na stacku. Kanárek se ukládá na pozici mezi zálohu registrů a vlastní lokální proměnné a jedná se o operaci xor mezi registry **EBP** a **ESP**. Toto přináší drobnou režii do výpočtu vlastního programu, neboť je třeba na konci volání funkce obsahující kanárka kontrolovat, zdali nebyl kanárek nějakým způsobem přepsán. Pokud se tak stane, vlastní program okamžitě vyvolá výjimka a skončí. Tato obrana ovšem není dokonalá a existují různé útoky, které ji překonávají.

2.7 Halda

Halda je druhý typ datové struktury velmi používané při psaní programů. Je primárně používána, pokud není dopředu jisté, jaká velikost paměti bude potřeba. Z tohoto důvodu není vlastní adresa paměti haldy již od počátku vlastněná programem. Ve skutečnosti je třeba spolupráce s operačním systémem, který musí pomocí MMU (memory management unit) předat tuto paměť (nějaké stránky paměti) programu, který ji požaduje. Toto je ovšem velmi zdlouhavá operace, protože vlastní operační systém, potažmo MMU musí nalézt vhodnou část paměti, kterou může předat. Navíc, protože tato paměť je přímo spravována programem, je třeba starat se o její dealokaci

(vrácení systému). Pokud není tato paměť vrácena, vezme si ji po ukončení běhu operační systém automaticky, ale to znamená, že program zbytečně používal zdroje operačního systému, kterých je omezené množství. Z tohoto důvodu je velmi důležité zacházet s touto pamětí velmi opatrně.

Kapitola 3

Možné zranitelnosti a útoky

Jedním z hlavních nebezpečí vývoje software je vytvoření nežádoucího chování programu. Takovéto chyby mohou být neškodné, ale zároveň mohou umožňovat útočníkovi zneužití programu v jeho prospěch. Pokud například vývojář software vytvoří program, který ukončí svůj výpočet hned na začátku bez toho, aniž by cokoliv udělal, jedná se samozřejmě o chybu programu. Takovouto chybu ovšem s velkou pravděpodobností nebude možné využít útočníkem. Pokud vývojář nechtěně vytvoří kvůli nepozornosti chybu s použitím nějaké funkce, které sám příliš dobře nerozumí, bude tuto chybu velmi obtížné nalézt a bude se jednat o vhodné místo pro útočníka, kde hledat útok, využívající této zranitelnosti. V následujících několika odstavcích budou shrnuty některé základní druhy útoků na programy s různými zranitelnostmi a způsob, jak se závažnost těchto zranitelností vyhodnocuje.

3.1 Kategorizace zranitelností

Ke kategorizaci zranitelností a útoků je možné využít několik různých způsobů. V této práci bude použita CVSSv2 (Common Vulnerability Scoring System), který je jeden z nejpoužívanějších systémů pro hlášení zranitelností díky své jednoduchosti. CVSSv2 se skládá ze tří základních skupin: Base Metrics, Temporal Metrics a Enviromental Metrics. Každá z těchto kategorií má své podkategorie, které budou představeny dále. Vlastní podkategorie jsou hodnoceny v rozmezí 0 až 10, kde 10 označuje velmi závažnou zranitelnost. V této práci bude použita pouze Base metrika, která ukazuje základní charakteristiky zranitelnosti, které jsou stejné nehledě na prostředí, ve kterém se zranitelnost vyskytuje.

Attack Vector (AV) Tato metrika určuje, jak může být zranitelnost využita. Jinými slovy, čím vzdálenější může být útočník od svého cíle, tím je větší hodnota této kategorie. Nejvyšších hodnot tedy budou dosahovat zranitelnosti, kterých je možné využít přes síť.

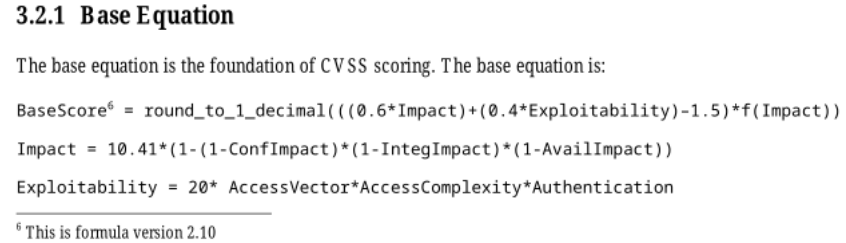
Access Complexity (AC) Složitost přístupu ke zranitelnosti určuje, jak je jednoduché zpřístupnit onu zranitelnost. Snadněji dostupné zranitelnosti zde dosahují vyššího skóre. Kupříkladu, pokud bude možné zaútočit na službu pouze 29. února, a to pouze pokud je útočník již v některé části systému, jedná se o velmi náročnou znanitelnost k zneužití.

Authentication (Au) Kategorie ukazující počet nutných přihlášení útočníka k oběti při pokusu o využití zranitelnosti. Čím více je nutných přihlášení, tím menší hodnotu má tato metrika.

Confidentiality Impact (C) Metrika určující dopad při úspěšném zneužití zranitelnosti. Jedná se o možnost útočníka získat informace, ke kterým by za běžných okolností neměl přístup. Kompletní přístup zde označuje schopnost číst data systému, a to přesně ta, o která žádal.

Integrity Impact (I) Kategorie umožňující zjistit, jak využití zranitelnosti změní integritu systému. Nejvyšší hodnoty kompromitace je možné docílit, pokud díky zranitelnosti útočník obejde veškerou obranu.

Availability Impact (A) Poslední metrika určující dopad využití zranitelnosti. Vše od spotřebované paměti, procesorového času nebo diskového prostoru patří do této kategorie.



3.2.1 Base Equation

The base equation is the foundation of CVSS scoring. The base equation is:

BaseScore[6] = round_to_1_decimal(((0.6*Impact)+(0.4*Exploitability)-1.5)*f(Impact))

Impact = 10.41*(1-(1-ConfImpact)*(1-IntegImpact)*(1-AvailImpact))

Exploitability = 20* AccessVector*AccessComplexity*Authentication

[6] This is formula version 2.10

Obrázek 3.1: Ukázka výpočtu závažnosti zranitelnosti podle CVSSv2[12]

Na předchozím obrázku 3.1 je vidět, jakým způsobem je vypočítána závažnost zranitelnosti při použití systému CVSSv2.

3.2 Odepření služby

Jednou z možných (a v jistém smyslu nejjednodušších) chyb v software je chyba umožňující odepření služby (Denial of service). Útok tohoto typu dovolí útočníkovi znehodnotit nějakou část systému oběti. Většinou je odepření služby spojováno se síťovými technologiemi kupříkladu ve formě distribuované verze tohoto útoku. Tento útok je ovšem možné uskutečnit i lokálně.

Na rozdíl od síťové verze, která většinou spoléhá na nedostatečné možnosti obránce obsloužit velké množství klientů, je lokální odepření služby postaveno spíše na technikálii. Jedna z možností je donutit program vypočítávat nekonečnou smyčku. Tímto se spotřebují zdroje systému, který bude alespoň zčásti ochromen. Další možnost je vyčerpání volné paměti systému. Tento druh útoku se velmi podobá síťovým verzím tohoto útoku, které jsou také většinou postavené na vyčerpání paměťového prostoru. V neposlední řadě je možné vynutit chování vytvářející různé hazardy při použití více vláken. Do této skupiny spadají všechny chyby vznikající při neopatrné manipulaci se sdílenými prostředky. Jedná se zejména o starvation, deadlock nebo livelock.

3.3 Přetečení bufferu

Přetečení bufferu (buffer overflow) je jedním z nejzákladnějších lokálních útoků. Tento útok spočívá v zapsání dat do části paměti a nekontrolování, zdali se nezapsalo více, než je velikost cílové paměti. Kupříkladu, pokud bude použita základní funkce `strcpy`, tak bude zkopírován první řetězec do druhého bez ohledu na velikost alokovaného pole. Jediné, co tato funkce řeší, je konec (nulový terminační znak) prvního řetězce. Tímto způsobem je ovšem možné přepsat nějakou část paměti. Vzhledem k tomu, že lokální proměnné jsou vytvořené na zásobníku při volání funkce, je možné zapsáním většího řetězce přepsat ostatní lokální proměnné nebo důležitá řídící data funkce. V některých případech je tedy možné přepsat návratovou hodnotu a donutit program vykonávat nezamýšlený kód [13].

3.4 Čtení neinicializované paměti

Ne všechny útoky mohou být na první pohled zřejmé jejich chováním jako DoS nebo přetečení bufferu. Zde není ani příliš jednoznačná terminologie, neboť se na problematiku čtení neinicializované paměti lze dívat jak na zranitelnost, tak na útok. Ve zkratce většinou tvůrce programu nechce, aby uživatel (nebo nějaká třetí strana) mohl pomocí tohoto programu scanovat vlastní program. Pokud by totiž v takovémto programu byly uloženy nějaké citlivé údaje, jako jsou například přihlašovací data či klíče, bylo by možné tyto informace z programu získat. Toto platí i pro programy, které téměř žadné takovéto informace neukládají. Pokud bude kupříkladu daný program používám jako nástroj, který sdílí více uživatelů, mohlo by být možné do jisté míry zjistit, co předchozí uživatel s tímto programem dělal. Toto je pouze jednoduchý příklad ukazující možné zneužití nějakého programu, který používá více než jeden uživatel. Zde se již ovšem dostáváme do specifikací operačních systémů, kde je v případě takovéto zranitelnosti velmi důležitá otázka, jak nakládá operační systém s hlavní pamětí. Je pamět mazána, pokud nějaký program dostane sekci, kterou dříve vlastnil jiný program, nebo se v rámci šetření předá

paměť v takovém formátu, v jakém byla odevzdána? Většina dnes používaných moderních operačních systémů (potažmo harwarových komponent) již toto chování vynucuje. Při vývoji software ovšem není možné (respektive nemělo by být) vynutit platformu, na jaké je program spouštěn. Je tedy žádoucí chovat se k paměti tak, jako kdyby na každé platformě toto chování nebylo vynuceno.

Kapitola 4

Rozbor programu

Při prvním pohledu na program IrfanView je možné předpokládat použití některých obran proti reverznímu inženýrství. K tomuto závěru jsem zpočátku došel kvůli použití různých funkcí, díky kterým se většinou ztěžuje analýza. Jmenovitě se jedná o funkce `GetTickCount`, `GetSystemTime` nebo instrukci RDTSC získávající přesný procesorový čas. Jak bylo psáno v teoretické části, tyto praktiky mohou svědčit o nějaké detekci debuggingu 1.4.5. Při důkladném prozkoumání této funkce je možné zjistit, že ač pozicí by odpovídala vhodnému místu pro obranu proti debugování, jedná se pouze o dodatečnou informaci o času stráveném nahráváním.

4.1 Postup analýzy

Při analýze programu IrfanView jsem se rozhodl pro hybridní přístup a použil jsem tedy jak metody statické analýzy, tak metody dynamické analýzy. Nejdříve jsem zmapoval zdroje programu z důvodu vytvoření určitého povědomí, jakým způsobem je program napsán. Následně jsem použil nástroj CFF Explorer, abych se ujistil, kdo vytvářel které části programu. Toto není nejlepší způsob detekce různých vývojářů, ale u tohoto programu byl tento přístup dostatečný. Díky tomuto přístupu jsem měl značné povědomí o programu ještě předtím, než jsem se do něj ponořil hlouběji.

Následně jsem již použil software jako je IDA nebo x64dbg a prohledával dissassemblovaný zdrojový kód. V této části jsem použil několik různých přístupů k rozboru. Jedním z těchto způsobů bylo získat z programu IDA všechny importované funkce a vytipovat z těchto funkcí ty, které mohou být špatně použity a způsobit tedy nějaký druh zranitelnosti. Následně jsem prohledal okolí těchto funkcí a ověřil si, že jsou správně ošetřeny. V některých případech se ovšem jednalo o velmi složitý úkol, a tak jsem program spustil s x64dbg a vytvořil obrázek ve formátu, díky kterému bylo možné identifikovat, zdali se jedná o zranitelnost. Tento přístup je asi nejefektivnější ze způsobů

analýzy, které jsem použil. Je ale možné, že při analýze uniknou některé zranitelnosti nezávislé na importovaných funkcích.

Z tohoto důvodu jsem použil méně efektivní způsob analýzy pro podezřelé části a ručně je odkrokoval. Tento přístup je extrémně náročný a pomalý, ale je možné tvrdit, že je preciznější. Bohužel jeho nevýhoda je rychlost prohledávání a nutnost dávat pozor na nejmenší detaily. Zde jsem kombinoval opět oba programy IDA a x64dbg. Občas jsem použil v této části i Process Hacker, pokud jsem potřeboval přesnější informace o aktuálním běhu programu.

Poslední použitý způsob je aplikovaný velmi často u různých CTF soutěží, a to je náhled na program z hlediska použitých řetězců. Tento způsob mi pomohl se orientovat v programu a dát si do perspektivy některé části programu. Sám o sobě ovšem nepřinesl mnoho.

4.2 Metadata

CFF Explorer nám dokáže prozradit několik zásadních informací o vlastním programu IrfanView. Z metadat je tedy možné zjistit, že se jedná o program IrfanView 64bitovou verzi pro Windows Vista, Windows 7, Windows 8 a Windows 10 ve verzi 4.58.0.0. Program je kompilován jako přenositelný spustitelný 64bitový soubor pomocí kompilátoru Microsoft Visual C++ 8.0, tato verze byla vytvořena 22. října 2021. Tyto informace jsou velmi důležité, neboť mohou prozradit, zdali pluginy k tomuto programu vytvořil stejný autor (Irfan Škiljan) nebo zdali jsou pluginy vytvořené někým jiným. K tomuto se také hodí znát Copyright, který je Copyright (co) 2021 by Irfan Skiljan, Austria.

4.3 Importy

IrfanView je program na prohlížení obrázků s vysokým stupněm modularity a možností přidání pluginů. Z tohoto důvodu má program dobrou podporu dynamicky linkovaných knihoven, takzvaných dll. Vlastní spustitelný soubor zpočátku linkuje šest dynamických knihoven. Tyto knihovny jsou COMCTL32.dll, což je knihovna použitá pro manipulaci s obrázky, statusy a nástrojovými lištami. Následně se linkuje jako téměř u každého programu KERNEL32.dll, který má za úkol importovat různé funkce, bez kterých se žádný program neobejde. Mezi tyto funkce patří kupříkladu `WriteFile`, `LoadLibraryExW`, `GlobalAlloc` a podobné. Při rozboru nutných knihoven je také potřeba zmínit knihovnu USER32. Tato knihovna importuje funkce, bez kterých se žádný program neobejde, podobně jako bez KERNEL32.dll. Na rozdíl ale od KERNEL32.dll jsou tyto funkce spojeny s userspace a obsahují tedy funkce jako `EnableMenuItem`, `GetKeyboardLayout` nebo například `LoadStringW`. Následuje knihovna GDI32.dll neboli Graphics Device Interface, která je nutná, neboť program obsahuje grafické rozhraní. Jedná se o knihovnu se základními grafickými funkcemi, jako je například `GetPixel`,

`CreateBitmap` a podobné. Poslední dvě knihovny importované po spuštění jsou ADVAPI32.dll a SHELL32.dll. ADVAPI32.dll je nutná knihovna, pokud program pracuje s registry (Registr Windows) a obsahuje tedy funkce jako `RegOpenKeyW` nebo `RegSetValueW`. Tyto funkce jsou velmi důležité, neboť program se jistí proti přemazání souborů při nahrávání tím, že takto načítané soubory zamyká přes systém registrů (zde je registrem myšlen systémový registr, nikoliv procesorový). SHELL32.dll obsahuje různé funkce dovolující otevírání souborů a webových stránek.

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
COMCTL32.dll	12	001F4E60	00000000	00000000	001F5C14	00182060
KERNEL32.dll	154	001F5098	00000000	00000000	001F6252	00182298
USER32.dll	166	001F55F8	00000000	00000000	001F6D30	001827F8
GDI32.dll	57	001F4EC8	00000000	00000000	001F70E6	001820C8
ADVAPI32.dll	11	001F4E00	00000000	00000000	001F7186	00182000
SHELL32.dll	16	001F5570	00000000	00000000	001F72D6	00182770

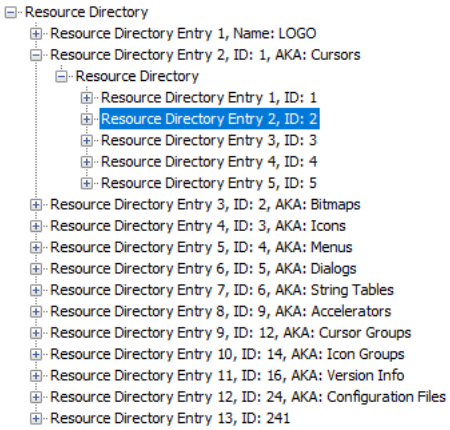
Obrázek 4.1: Ukázka výpisu z programu CFF Explorer při pohledu na seznam knihoven.

V průběhu se načtou ještě tři další systémové knihovny SHLWAPI.dll, COMDLG32.dll a OLE32.dll. SHLWAPI.dll u programu IrfanView slouží primárně k importu funkce `StrCmpLogicalW`, která slouží k porovnání dvou unicodových řetězců bez ohledu na velikost písmen. COMDLG32.dll importuje různé funkce umožňující jednodušší práci se soubory a fonty. Velmi často používaná funkce z této knihovny je `GetOpenFileNameW`, která se v programu IrfanView používá primárně k opětovnému získání jména otevřeného souboru. V neposlední řadě se importuje systémová funkce OLE32.dll sloužící k přidání různých nadstandardních vlastností celému programu. Mezi tyto vlastnosti patří primárně `DoDragDrop` funkce umožňující načíst soubor pouze jeho přetažením do aktivního okna programu.

4.4 Zdroje

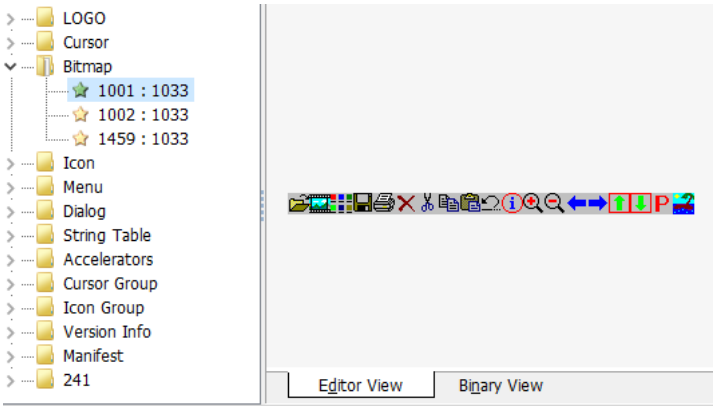
Pro nalezení zdrojů je možné použít program CFF Explorer, který dokáže identifikovat různé zdroje, které program využívá a které jsou uvnitř uloženy. Pro jednoduché zobrazení je ale vhodnější nástroj jako Resource Hacker, který nám dovolí jednoduše procházet různé zdroje daného programu. Zdroje se dělí do třinácti skupin. V této práci nebude rozebrána každá z těchto skupin, ale pouze ty nejdůležitější z nich. Velmi důležitá skupina zdrojů je jednoznačně skupina kurzorů. Jedná se o 5 nestandardních kurzorů, které program IrfanView v určitých momentech využívá. Je důležité poznamenat, že toto

nejsou jediné kurzory, které se mohou vyskytovat při práci s tímto programem. Většinou se totiž pracuje pouze se systémem definovanými kurzory.



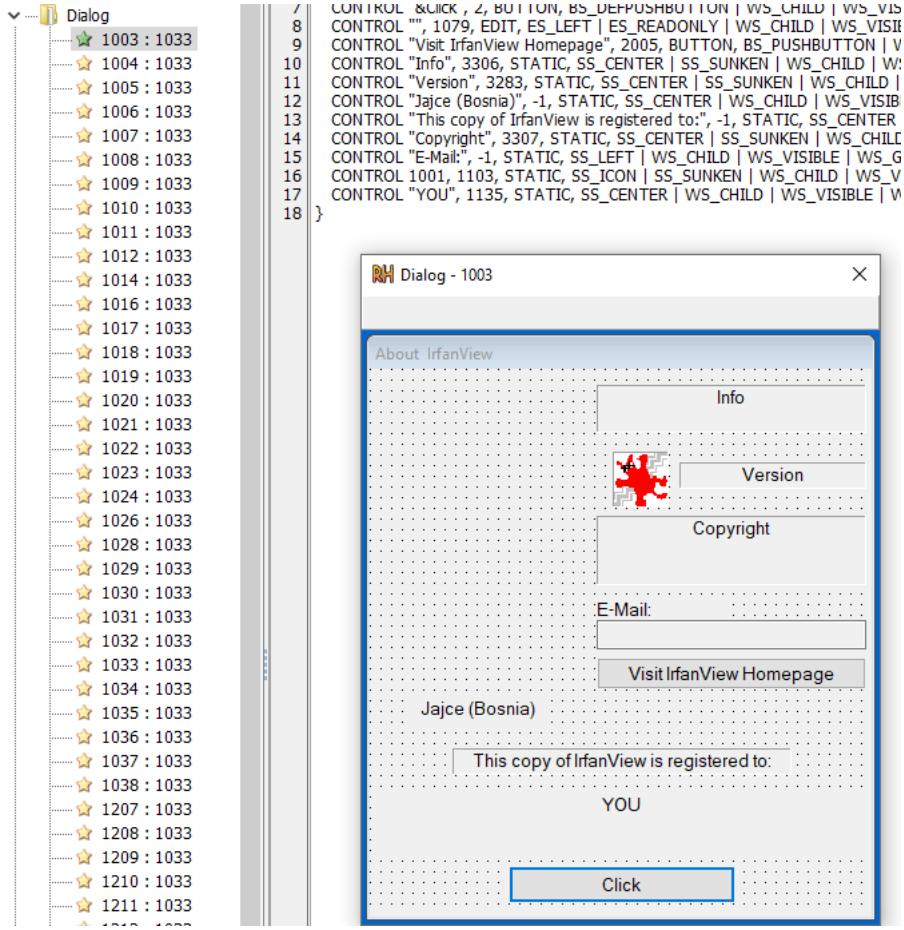
Obrázek 4.2: Seznam nestandardních kurzorů používaných programem IrfanView.

Sekce Icon a Bitmap obsahují různé obrázky nástrojů. Kupříkladu celá nástrojová lišta je jedna bitmapa, která je pouze škálována podle požadavku uživatele. Toto je naprosto standardní způsob, jak vytvořit nástrojovou lištu. K této skupině ještě přidám sekci Menu, která obsahuje různé vzory horní lišty.



Obrázek 4.3: Ukázka uložení nástrojové lišty pomocí programu Resource Hacker.

Velmi obsáhlá sekce je sekce Dialog. V této sekci je možné nalézt různé vzory dialogových oken. Ve skutečnosti by bylo velmi náročné pokaždé při použití dialogového okna znovu jej vytvářet. Nejsou zde ovšem všechna možná dialogová okna, pouze ta často používaná. Dohromady je možné v programu nalézt 70 různých vzorů dialogových oken.

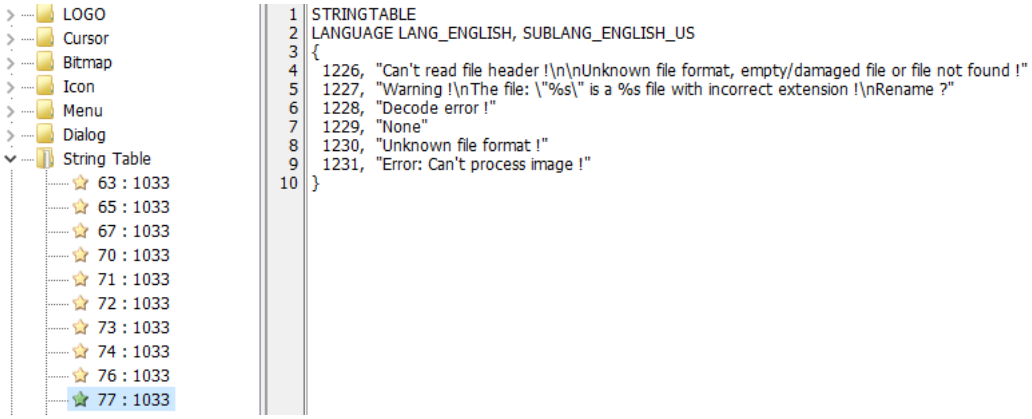


Obrázek 4.4: Ukázka často používaných dialogových oken. Na obrázku je možné vidět vzor pro sekci O programu.

Poslední velmi důležitou skupinou zdrojů je String Table. Jak již název napovídá, vyskytují se zde různé znakové řetězce používané programem. Nejsou zde všechny, ale opět stejně jako u dialogových vzorů pouze ty nejvíce používané. Ve skutečnosti se ovšem nejedná pouze o řetězce, ale o skupiny řetězců. Některé nejsou příliš zajímavé, jako například 63:1033, který obsahuje pouze slovo Clipboard. Vyskytují se zde ovšem i skupiny obsahující velmi důležité řetězce, jako je například 77:1033, ve které se nacházejí různé chybové hlášky spojené s načítáním souborů.

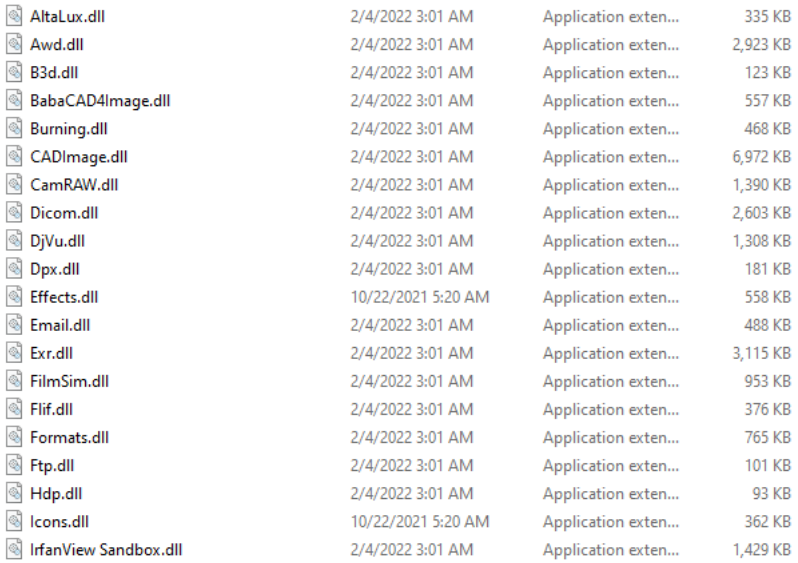
4.5 Importované pluginy

Jak již bylo naznačeno v teoretické části, některé programy velmi závisejí na využití různých pluginů. Ani program IrfanView není výjimkou, a má tedy velkou podporu pluginů. V této analýze se zaměřím pouze na pluginy, které



Obrázek 4.5: Skupina uložených často používaných řetězců.

jsou oficiálně podporované tvůrcem programu. Jinými slovy, pokud je možné nainstalovat plugin pomocí oficiální instalace programu IrfanView, jedná se o oficiálně podporovanou součást programu. Z velké části jsou pluginy vytvářené Irfanem Škiljanem. Většina pluginů popsaných v této práci jsou pluginy 64bitové verze programu IrfanView.



Obrázek 4.6: Náhled do adresáře obsahujícího různé knihovny.

AltaLux.dll Altalux.dll je knihovna implementující takzvaný AltaLux filter pro program IrfanView, který by měl vylepšit kvalitu aktuálního zobrazeného obrázku ve formátu bitmap. Jejím autorem je Stefano Tommesani (Copyright Stefano Tommesani 2005/2015) a jedná se o open source plugin, který je možné nalézt na Github stránce autora StefanoT [14]. Jedná se pravděpodobně o

opuštěný nebo málo žádaný plugin vzhledem k tomu, že neobsahuje žádnou historii reportovaných chyb a poslední update tohoto projektu byl před téměř čtyřmi lety. Ve skutečnosti tento uživatel má poslední Contribution do svých projektů právě do této knihovny 5. března v roce 2018.

Awd.dll Teto plugin přidává podporu pro Artweaver formát souboru. Tento plugin byl vytvořen společností Boris Eyrich Software původem z Německa a jedná se primárně o demo verzi hlavního produktu firmy. Tímto produktem je Artweaver vytvořený v roce 2009. Verze pluginu používaná v programu IrfanView je verze 2.0.0.0 [15].

B3d.dll Plugin bez dodatečných informací na zobrazení b3d souborů. Je důležité zmínit, že tento plugin nemá nic společného s Blitz3D engine, který označuje své soubory .b3d. Ve skutečnosti se zde jedná o metodu zobrazení souborů s formátem Ben's 3D Format. Původně byl tento formát vytvořen pro použití s OpenGL Performer knihovnou pro vytváření programů pro virtuální realitu.

BabaCAD4Image.dll Knihovna je napsaná vývojářem Miraza Coralic a jedná se o plugin na bázi programu BabaCAD. Jedná se tedy o knihovnu zprostředkovávající konverze formátů obrázků vytvářených programem babaCAD, což je program na vytváření a prohlížení technických výkresů. Jmenovitě tato knihovna přidává podporu souborů typu DXF a DWG [16].

Burning.dll Burning je knihovna napsaná původním autorem programu IrfanView (Irfan Škiljan). Jejím hlavním úkolem je rozšíření programu IrfanView, aby dokázal snadno vypalovat různé prezentace na optické disky.

CADImage.dll Knihovna pochází od firmy Soft Gold založené v roce 2000 v Rusku s webovou stránkou cadsofttools.com. Při nainstalování tohoto pluginu není třeba používat plugin BabaCAD4Image, protože CADImage dokáže také přečíst soubory typu DXF a DWG. K tomu je možné tento plugin použít na čtení HPGL2, CGM a SVG.

CamRAW.dll Další plugin od vývojáře Irfana Škiljana, tentokráte na čtení raw formátů souborů z fotoaparátů. Podporuje velkou řadu formátů, mezi které patří např. mimo jiné DNG, ORF, RAF, MRW nebo X3F.

Dicom.dll Díky DICOM pluginu je možné v programu IrfanView prohlížet formáty, jako jsou například DCM, ACR nebo IMA. Tyto formáty jsou primárně využité ve zdrovotnictví a dokáží zobrazovat různé soubory ze scanerů. Celá zkratka DICOM je poté Digital Imaging and Communications in Medicine.

DjVu.dll Plugin na zobrazení souboru ve formátu DJVU — opět se jedná o plugin vytvořený Irfanem Škiljanem, který je primárně zaměřen na prohlížení různých dokumentů. Tento formát vytvořený v AT&T je alternativou pro již dnes standardní PDF formát. Navíc na rozdíl od předchozích pluginů od stejného autora se na tomto pluginu podílela i firma Caminova zabývající se vývojem nástrojů pro úpravu grafických souborů.

Dpx.dll Umožňuje zobrazení souborů ve formátu DPX a CIN a jejich omezenou úpravu. Jedná se totiž primárně o jednotlivé rastrové snímky filmu. Tento plugin byl stejně jako většina ostatních vytvořen Irfanem Škiljanem.

Effects.dll Plugin umožňující použití různých filtrů z programu Adobe Photoshop. Tvůrcem je opět Irfan Škiljan.

Email.dll Plugin přidávající možnost odeslat obrázek pomocí emailu. Nejedná se ovšem o klasické volání základního emailového klienta, ale přímo zabudovaný emailový klient. Stejně jako u předchozího pluginu je autorem Irfan Škiljan.

Exr.dll Tato knihovna přidává podporu čtení EXR (OpenEXR High Dynamic-Range bitmap file) souborů. Autorem je Irfan Škiljan.

FilmSim.dll Tento plugin umoňuje použití různych filmových efektů. Vývojář tohoto pluginu (Jan Ingwer Baer) zde pravděpodobně udělal drobnou chybu při zapisovaní verze této knihovny, kde místo standardních teček používá čárky. Jedná se o velmi nový plugin, který byl přidán až ve verzi 4.44 [17].

Flif.dll Další plugin vytvořený Irfanem Škiljanem je tentokrát určený na četbu souborů typu FLIF. Jedná se o zkratku z anglického Free Lossless Image.

Formats.dll Plugin opět vytvořený Irfanem Škiljanem na čtení různých formátů – nejedná se o příliš důležitý plugin z toho důvodu, že ač přidává velké množství formátů, které může číst, jedná se spíše o formáty některých starých souborů, jako formáty z dob ZX Spectrum, Amigy a Atari. Kromě těchto zastaralých formátů čte tato knihovna různé málo používané formáty souborů. Z tohoto důvodu lze předpokládat, že se v této knihovně bude vyskytovat velké množství zranitelností, což potvrzují i stránky udržující přehled o zranitelnostech, jako je například cvedetails.com.

Ftp.dll Ftp plugin umožňuje přesun a stažení různých thumbnailů za pomoci FTP protokolu. Jedná se o velmi malý plugin s poměrně rozsáhlým přístupem k internetu. Plugin je opět vytovřen Irfanem Škiljanem.

Hdp.dll Další velmi malý plugin, který přidává možnost četby formátů HDP a WDP, navíc rozšiřuje četbu JPEG souborů o formát JPED-XR. Při zanedbání knihovních funkcí se jedná o velmi malý plugin spíše využívající různé naprogramované funkce z knihoven jako ole32.dll nebo WindowsCodecs.dll vytvořený Irfanem Škiljanem.

Icons.dll Dle stránky popisující pluginy IrfanView tato knihovna přidává různé ikony pro soubory asociované s programem IrfanView. Tento plugin je ovšem velmi podezřelou součástí plugin sady, neboť se jedná o plugin, který nebyl vytvořen autorem původního programu, ale Florianem Kilzerem. Toto jméno je spojeno s univerzitou TU Wien, Vídeň. Bohužel se mi nepodařilo dohledat, zdali tento plugin skutečně vytvořil, zdali se jedná o jmenovce nebo přímo podvržené jméno, které bylo náhodně vybráno pro zvýšení důvěryhodnosti. Tato obezřetnost není neopodstatněná, neboť se jedná o packovaný binární soubor, není tedy příliš snadné do něho nahlédnout. Celý plugin je vytvořen s licencí Creatice commons Attribution-ShareAlike 2.5 [18].

Sandbox.dll Přidává rozšíření umožňující použití JewelScript effektů a filtrů. Plugin je vytvořen Stefanem Kuhnem v roce 2015.

Jpeg_LS.dll Plugin přidávající podporu čtení a zápisu JPEG-LS neboli Lossless JPEG – vývojářem tohoto pluginu je Kanryu Kato, který na vytvoření tohoto pluginu spolupracoval s Irfanem Škiljanem. Plugin je vytvořen na základu charls knihovny, pro kterou Kato Kanryu vytvořil původní verzi barevné transformace [19].

JPEG2000.dll Plugin vytvořený Irfanem Škiljanem jako další rozšíření pro čtení formátu JPEG – vlastní implementace je vytvořena na základě SDK knihovny od firmy LuraTech GmbH pro čtení JPEG ve formátu JPEG 2000.

JpegQS.dll Plugin umožňující čtení a vytváření JPEG souborů metodou QuantumSmooth – vlastní plugin není nijak podepsán, ale je možné zjistit ze struktury pluginu, že tvůrcem je uživatel ilyakurdyukov. Vlastní zdrojový kód pluginu je veřejně dostupný na githubu [20].

Jpg_transform.dll Přidává podporu čtení a zápisu JPEG souborů vytvořené Irfanem Skijanem. Jedná se o plugin dovolující různé bezztrátové transformace souboru typu JPEG. Mezi přidané transformace patří Cropping, rotace nebo například editace EXIF metadat u JPG typu souboru.

JPM.dll Další rozšíření určené pro čtení rozšiřujícího formátu JPEG zvaného JPM, celým názvem se jedná o JPEG 2000 Multi-layer bitmap file. Tento plugin, stejně jako většinu ostatních na čtení JPEG formátů a jeho

derivátů, vytvořil Irfan Škiljan. Tento plugin je stejně jako JPEG2000.dll vytvořen ze základu SDK knihovny od firmy LuraTech GmbH.

Lcms.dll Lcms.dll přidává možnost použití barevných filtrů (v oficiálních dokumentech se tato vlastnost nazývá Color Management). Tento filter byl vytvořen Irfanem Škiljanem.

Metadata.dll Je jedno z novějších rozšíření programu IrfanView přidané Irfanem Škiljanem v roce 2021. Díky tomuto pluginu je možné číst a editovat různá metadata formátu, jako je například EXIF nebo IPTC. Přesněji se jedná primárně o četbu komentářů, které je možné vložit do metadat těchto souborů.

Mng.dll Plugin umožňující čtení souborů MNG a JNG – jedná se o rozšíření formátu PNG vytvořené Irfanem Škiljanem. Soubor typu MNG je ve skutečnosti pouze několik PNG obrázků a jedná se tedy o animovaný formát. Celý název tohoto formátu je Multiple-image Network Graphics. Druhý formát, který je možný díky tomuto pluginu číst, je JNG, což je JPEG Network Graphics bitmap file.

MrSID.dll Multiresolution seamless image database nebo také MrSID je formát obrázku vytvořený společností TechLizard. Vlastní plugin je závislý na knihovnách lti_dsdk_cdll_9.0.dll a tbb.dll, obě jsou vytvořené firmou TechLizard. Autorem knihovny MrSID.dll je ovšem Irfan Škiljan.

OptiPNG.dll Malý plugin umožňující optimalizované ukládání formátu PNG. Tento plugin byl napsán Irfanem Škiljanem a jeho poslední verze 4.58.0.0 je z roku 2021.

Paint.dll Jednoduché rozšíření programu IrfanView přidávající možnosti drobné grafické editace – mezi tuto editaci patří různé malování čar, základních geometrických tvarů a drobná úprava editovaných obrázků. Plugin byl vytvořen vývojářem Matteo Italia. Vývojář také vytvořil vlastní webovou stránku s dokumentací k tomuto pluginu [21].

PDF.dll Jak již název rozšíření naznačuje, jedná se o knihovnu umožňující čtení a vytváření souborů typu pdf. Vývojářem, stejně jako u ostatních základních komponent programu IrfanView, je Irfan Škiljan.

Postscript.dll Je plugin umožňující čtení souborů typu EPS, PS a PDF pomocí GhosScriptu. GhostScript je interpreter pro Postscript používaní v produktech společnosti Adobe. Plugin napsal Irfan Škiljan. Tento plugin navíc nebyl překompilován do verze 4.58.0.0 (Jeho verze je 4.57.0.0), což je běžné u pluginů vytvořených Irfanem Škiljanem.

RegionCapture.dll Plugin umožňující vytvářet screenshoty specifické části obrazovky. Jedná se o podobnou funkcionalitu, kterou má SnippingTool, který je součástí nových instalací operačního systému Windows. Možná také kvůli této skutečnosti není tento plugin příliš updatovaný. Poslední verze je 2.4.3 z roku 2009. Vývojářem je Itay Szekely vystupující pod přezdívkou grebulon. Plugin je inspirován programem DCUtility. Na webové stránce pluginu [22] je možné nalézt i drobný popis tohoto pluginu stejně jako zdrojový kód celého pluginu.

Sff.dll Plugin umožňující čtení souborů typu SFF neboli Structured Fax File – vlastní plugin nemá v metadatech napsaného autora a nepodařilo se mi ho dohledat.

SVG.dll Plugin vytvořený vývojářem Seppe Sol pro prohlížení a editaci souborů ve formátu SVG. Tento plugin ovšem není příliš vhodný na editaci SVG souborů, neboť nepracuje příliš dobře s vektorovým obrázkem. Hlavní problém je, že při přiblížení se vlastní obrázek nezaostří, což je limitací vlastního pluginu [23].

Tools.dll Implementuje některé dodatečné nástroje do programu IrfanView, jako jsou kupříkladu různé textové efekty a titulky. Navíc rozšiřuje počet formátů, které je možné číst, o formáty HEIC a dle specifikací AVIF. HEIC je poměrně složitý formát, ale ve zkratce se jedná o aplikaci často používaného kodeku HEVC (High Efficiency Video Coding neboli h.265) na soubor typu HEIF, což je kontejner, který obsahuje různé obrázky v různých formátech. Tento nástroj byl vytvořen Irfanem Škiljanem.

Video.dll Plugin vytvořený Irfanem Škiljanem pro čtení video formátů. Program IrfanView není ovšem primárně k tomuto úkonu určený a jedná se tedy spíše o drobné rozšíření. Mezi podporované formáty patří kupříkladu formát AVI.

WebP.dll WebP neboli Weppy je formát souboru vytvořený společností Google, který umožňuje bezztrátovou i ztrátovou kompresi obrázku. Jedná se o pokus nahradit formáty JPEG, PNG a GIF. Tento plugin byl vytvořen Irfanem Škiljanem a jedná se opět o volitelnou součást programu, neboť je kompilovaný pro nižší verzi programu IrfanView.

WPG.dll Nový plugin přidaný ve verzi 4.57.0.0 v lednu roku 2021. Tento plugin slouží k četbě souborů ve formátu WPG neboli WordPerfect Graphics.

Wsq.dll Plugin umožňující četbu souborů ve formátu WSQ neboli Wavelet Scaler Quantization File. Vlastní plugin není příliš dobře podepsán, je možné pouze zjistit, že byl tento plugin vytvořen na konci roku 2019.

Xcf.dll Poslední plugin dovoluje čtení XCF souborů. Tento druh souborů bývá většinou spojen s programem GIMP. Plugin byl vytvořen vývojářem jménem Jacek Sobolewski.

4.6 Potencionálně nebezpečné funkce

V programu IrfanView je možné nalézt velké množství funkcí, které při špatném použití lze považovat za velmi nebezpečné. V programu IrfanView jsou všechny funkce pracující s Windows registry dobře ošetřené. Jedná se o funkce, jako je například `RegSetValueW` nebo `RegDeleteValueW`. V celém programu ani v jednom z pluginů není možné tyto funkce zneužít kupříkladu k vytvoření a nesmazání klíče v registru Windows nebo mazání cizího klíče.

Jeden z dobrých kandidárů na hledání zranitelností jsou funkce kopírující řetězec do jiného řetězce nebo proudu. Nejzákladnějšími takovými funkcemi jsou funkce `scanf` a `printf`. Tyto funkce v nejzákladnější formě ovšem není možné v programu nalézt, neboť řetězce nejsou nikdy vypisovány na standardní výstup/vstup. Naopak je zde možné nalézt funkce, jako je například funkce `sprintf`, `fprintf` nebo `sscanf`. Všechny tyto funkce jsou správně ošetřeny a z jejich použití neplyne žádná zranitelnost. Za zmínku stojí i importovaná funkce `wsprintf` kopírující data do předpřipraveného bufferu. Tato funkce je ale omezená na maximální velikost zapisovaného řetězce 1024 bytů. Na všech pozicích, kde se tato funkce používá, je vytvořen dostatečně velký buffer na pojmutí celého řetězce.

Funkce `fread` je funkce umožňující čtení bytů ze souboru. Z tohoto důvodu potřebuje funkce fread dostatečně velký buffer, aby nemohla vytvořit jakýkoliv druh přetečení. Návratová hodnota této funkce navíc umožňuje kontrolovat počet přečtených bytů. Kontrola počtu načtených bytů je ale v programu IrfanView spíše vzácný úkaz. Ve většině případů se ale nejedná o problém z hlediska bezpečnosti, protože to nevytváří žádný druh zranitelnosti.

`GetDlgItemText` je funkcí získávající text z různých uživatelem editovatelných částí dialogových oken. Z tohoto důvodu se jedná o ideálního kandidáta na prozkoumání, zdali mají buffery dostatečnou velikost a neumožňují třeba přetečení bufferu. Ve všech případech je tato funkce správně ošetřena a tuto zranitelnost neumožňuje.

Další kandidát na hledání možné chyby byla funkce `LockResource`. Při nedostatečném ošetření by mohlo být možné uzamknout nějaký zdroj, který by nebylo možné znovu uzavřít a případně získat. Toto by teoreticky mohlo způsobit předčasné ukončení programu. Ve všech instancích je ovšem tato funkce správně ošetřena a vždy uvolňuje tento zámek.

Velmi zajímavá funkce je `GlobalAlloc`. Tuto funkci používá program velmi často na alokaci. Důvod zkoumání této funkce je její možnost neinicializovat alokovanou paměť. Tato paměť by poté mohla být eventuelně přečtena a jednalo by se tedy o zranitelnost. Funkce `GlobalAlloc` je ale ve všech částech programu doprovázena hodnotou 0x40 na pozici řídícího příznaku. Díky této hodnotě příznaku je paměť vždy nulována. Další způsob, jak využít této funkce, je při špatné detekci nemožné alokace. I toto chování program ve všech případech detekuje a končí s předvídatelnou chybou.

Funkce pracující s řetězci jsou častým zdrojem chyb typu přetečení bufferu. Z tohoto důvodu jsem se zaměřil na dvojici funkcí `WideCharToMultiByte` a `MultiByteToWideChar`. V celém programu ani jedna z těchto funkcí není špatně napsána a vždy jsou správně ošetřeny krajní možnosti. Tyto krajní možnosti často velmi úzce souvisejí s velikostí alokované paměti obsahující/získávající řetězec znaků.

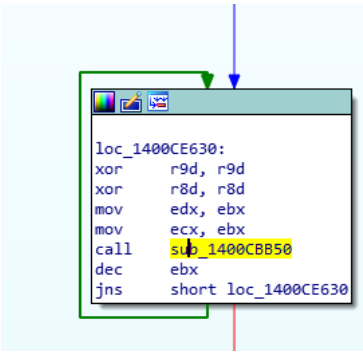
Všechny funkce typu `CreateToolbarEx` jsou správně ošetřeny a nemůže se tedy stát, že se spustí program bez nástrojové lišty. Spuštění bez nástrojové listy by velmi omezovalo možnosti práce programu a jednalo by se tedy o určitý druh DOS.

Funkce z knihovny COMCTL.dll jsou správně ošetřené a nevytvářejí dodatečné zranitelnosti v programu IrfanView. Většinou se zde jedná o funkce manipulující s obrázky, bylo tedy možné předpokládat, že při použití těchto funkcí bude špatně ošetřený některý z argumentů nebo návratová hodnota. Takováto zranitelnost by měla velký dopad na bezpečnost programu, protože tato třída má na starosti manipulaci s více obrázky. V programu IrfanView jsou tyto funkce použity primárně na manipulaci s obrázky rozhraní (ikonami nástrojové lišty apod.).

Při vlastní analýze mě zaujala funkce `GetTickCount`, která může být využita pro detekci debuggeru. V programu IrfanView je tato funkce použita pouze na několika místech s různým záměrem. Při prvním pohledu je možné se domnívat, že jedno z těchto použití je jasný případ detekce debuggeru, neboť program zavolá tuto funkci, uloží výsledek a pokračuje. Následně zavolá velkou funkci a při návratu z této funkce opět zavolá `GetTickCount` a rozdíl těchto hodnot uloží do paměti. Ve skutečnosti se ovšem jedná o informaci poskytovanou uživateli o době načtení obrázku (spuštění funkce). Druhé použití je jako seed pro funkci `srand`.

Další kandidát na potencionální DoS útok byla funkce `GlobalLock`. Tuto funkci je možné využít velmi podobně jako funkci `LockResource`. Stejně jako funkce `LockResource` je ale funkce `GlobalLock` dobře ošetřena z hlediska návratové hodnoty a jejího protikladu `GlobalUnlock`.

Funkce `SetCursorPos` by mohla způsobit velmi efektivní DoS, kdyby byla součástí nekonečného cyklu. V takovém případě by mohla způsobit uzamčení kurzoru na nějaké pozici a nemožnost s ním pohybovat. Tato funkce je v cyklu volána pouze ve dvou případech přímo ve funkci `WndProc`. Ani v jednom případě ale není možné vytvořit nekonečnou smyčku, protože řídícím registrem



Obrázek 4.7: Cyklus volající funkci obsahující SetCursorPos.

je registr **RBX**, který je pouze zmenšován v hlavním těle cyklu 4.7.

V předchozích odstavcích bylo představeno několik funkcí, na které jsem se v analýze zaměřil. Nejedná se ale o všechny funkce. Mezi další zkoumané patří například funkce, jako jsou `PeekMessageW` nebo `LoadStringW`. Ve většině případů jsou všechny použité funkce velmi dobře ošetřeny proti různým zranitelnostem.

Kapitola 5

Zranitelnosti programu

I když stránka cvedetails.com programu IrfanView vypadá tak, jak by bylo možné očekávat u takto starého programu, jedná se pravděpodobně o velmi dobře udržovaný software. Vývojář reaguje na většinu nalezených chyb do týdne, pokud se nejedná o složitější chyby. Je zde také důležité zmínit, že ne všechny nahlášené chyby budou v následujících sekcích diskutovány. `cvedetails.com` nerozlišuje mezi 32 a 64bitovou verzí programu IrfanView a může se stát, že jsou některé chyby architekturně závislé. Stejně tak je možné, že nalezené/analyzované chyby v 64bitových komponentách programu nemusejí být aplikovatelné na 32bitové verze. Jmenovitě z důvodu nekompatibility 64bitových knihoven pro 32bitové programy nemusejí být tyto zranitelnosti „přenositelné“. Navíc velká část zranitelností se objevila v minulosti, a je tedy možné, že některé části analýzy nemusejí být stoprocentně přesné. Díky rychlým reakcím autora je téměř nemožné získat postižené verze programu a přesně tedy určit chybu a její opravu.

5.1 Analýza oprav

Jak bylo v předchozích sekcích psáno, program IrfanView není zcela psán pouze jedním člověkem. Části programu, které spravuje Irfan Škiljan mají většinou jednotnou verzi a je tedy možné předpokládat, že při vytvoření nové verze programu jsou upraveny všechny zdrojové soubory v nové verzi. Některé části jsou ovšem vyvíjeny externě různými dalšími entitami. Z tohoto důvodu je občas třeba určit specifickou verzi pluginu, ve kterém se chyba nachází. Všechny chyby analyzované v této sekci je možné nalézt na webové stránce cvedetails.com. Při zkoumání oprav se mi navíc podařilo nalézt github stránky, obsahující některé chybové vstupní soubory pro důvody testování[24] [25] [26] [27].

V následujících několika kapitolách budou nejdříve zhodnoceny předcházející zranitelnosti a jejich opravy. Následně budou rozebrány nalezené nové nedostatky a zranitelnosti programu IrfanView. Postup byl zvolen

tak, aby bylo možné nejdříve zjistit, jaký druh zranitelností se v programu primárně vyskytuje a jak na ně autor reaguje. Díky tomuto postupu je možné se lépe zaměřit na nalezení některých druhů chyb a zranitelností.

5.1.1 Ostatní zranitelnosti

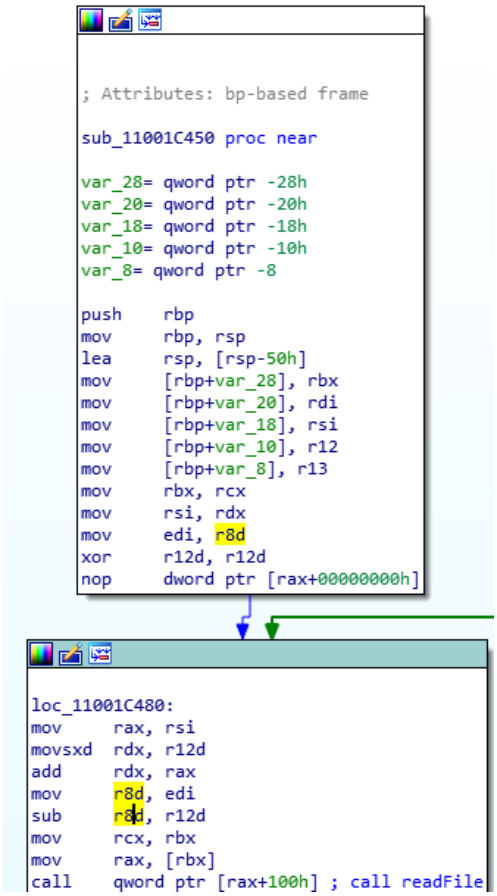
Vzhledem ke stáří programu se v něm v minulosti vyskytovalo mnoho zranitelností. Jmenovitě se jedná o 157 zveřejněných zranitelností v průběhu přibližně 25 let [28]. V této práci nejsou diskutovány všechny zranitelnosti, které se v minulosti v tomto programu vyskytly, ale pouze několik z nich. Ve skutečnosti je velmi obtížné analyzovat některé zranitelnosti vzhledem k aktivnímu vývoji programu a rychlým reakcím na nahlášené zranitelnosti. Čím více jsem se pokoušel analyzovat chyby z minulosti, tím bylo jasnější, že u takto dynamicky měnícího se programu bude téměř nemožné tyto zranitelnosti, případně jejich opravy nalézt. Z tohoto důvodu jsem se rozhodl analyzovat pouze chyby z roku 2020 a mladší. Jedná se poněkud o arbitrální číslo, které ovšem nebylo vymyšleno čistě bez rozmyslu. V tomto rozpětí je totiž možné pozorovat veškeré druhy zranitelností, a tedy i schopnost autora na tyto zranitelnosti reagovat.

5.1.2 Exec Code Overflow

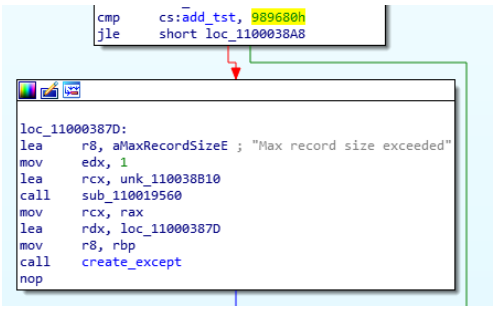
Tato chyba spočívá v přepsání nějakého interního pole přes jeho originální velikost. Přepsání kupříkladu návratové adresy funkce je možné spustit vlastní software.

CVE-2021-29367 Zranitelnost publikovaná 28. 9. 2021 ukazuje na možné přetečení při načítání souboru typu WPG. Tato chyba se vyskytuje v pluginu WPG.dll v3.1.1.0. Spočívala v upravení načítaného souboru takovým způsobem, aby při načtení vypadal soubor kratší, než ve skutečnosti je, a tím pádem naalokoval příliš malou paměť. Tato chyba byla ovšem velmi rychle opravena, a to ve verzi 3.1.2.0, kde je správně detekována velikost souboru a program odmítne načíst soubor se zobrazením chyby. Jak je vidět na obrázku 5.2, autor tuto zranitelnost opravil přidáním limitace na maximální velikost načítaného bloku. Není tedy možné načíst více než 10 MB dat v jednom bloku.

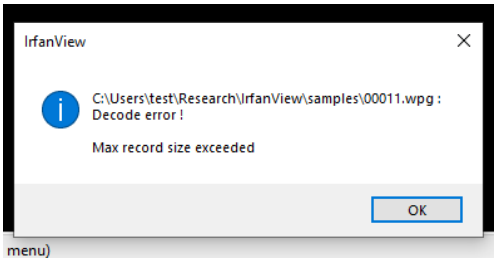
CVE-2021-29366 Zranitelnost z 28. 9. 2021 využívající špatné velikosti bufferu při použití funkce GetTempPathA. Tato funkce, jak již název napovídá, načte dočasný soubor. Z dokumentace je možné se dozvědět, že tato funkce dokáže přečíst maximálně 260 znaků plus znak koncový. V programu je napevno použita hodnota 0x104 při načítání dat označující maximální možnou velikost načtených dat. Oprava se zde vyskytuje ve formě zvětšené proměnné, do které se mají načítat data. Nově má buffer 0x410 bytů, což je více než dostatečná velikost.



Obrázek 5.1: Čtení souboru způsobující pokus o přetečení.



Obrázek 5.2: Větev programu detekující pokus o přetečení.

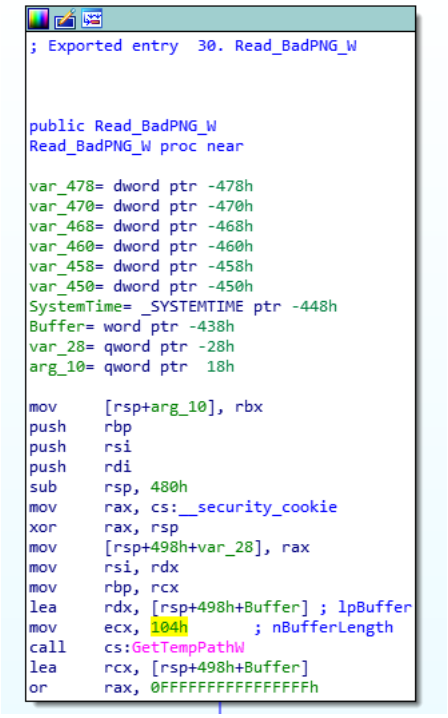


Obrázek 5.3: Chybové hlášení u chyby cve-2021-29367.

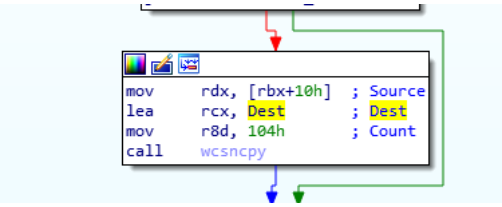
CVE-2021-29364 Jedná se o zranitelnost související se špatným ošetřením velikosti při kopírování cesty k souboru. Nově k tomuto úkolu použil autor funkci wcsncpy, která umožňuje omezit maximální velikost kopírovaného textu. Jak je na obrázku 5.5 vidět, velikost je omezena na maximální možnou velikost cesty v operačním systému Windows, tedy 260 znaků[29].

Zde je vhodné zmínit, že ani použití dlouhé cesty k souboru zde nevyvolá neočekávané chování. Program IrfanView vždy pracuje s dočasnými soubory. Pokud je tedy vytvořen soubor s větší délkou cesty než 260 znaků, je vytvořen dočasný soubor, který je efektivně kopií původního souboru. Název tohoto dočasného soubor s velmi krátkou cestou je předáván přes všechny části programu a s originálním souborem se vůbec nepracuje. Z tohoto důvodu není možné vytvořit přetečení pomocí dlouhé cesty k souboru.

CVE-2021-29363 Tuto zranitelnost z 28. 9. 2021 je velmi obtížné nalézt, ale jako většina zranitelností v hlavičce formats ukazuje na funkci, která má na



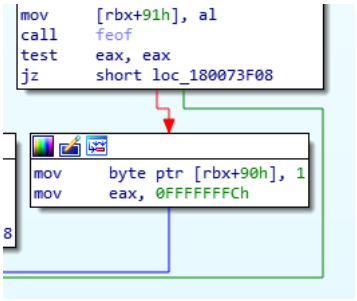
Obrázek 5.4: Ukázka opravy zranitelnosti cve_2021_29366.



Obrázek 5.5: Ukázka opravy zranitelnosti cve_2021_29364.

starosti čtení souboru. Zobrazení chybové zprávy se rozhoduje na konci funkce `open_RLE_head`. Zde se provede pokus o přeskočení na konec souboru dle specifikací formátu a následně se testuje feof. Tato chyba je správně ošetřena a není zde možné přetečení a následné přepsání návratové adresy.

CVE-2021-29362 Tuto zranitelnost, stejně jako předchozí, je velice obtížné nalézt z důvodu radikálních změn ve zdrojovém kódu. Je ovšem možné se domnívat dle popisu zranitelnosti a přiloženého testovacícho souboru, že se jedná o podobnou chybu, jako byla chyba předchozí. Tentokrát se ovšem jedná o špatný formát, který by mohl program donutit ke čtení více dat, než je naalokovaný prostor.



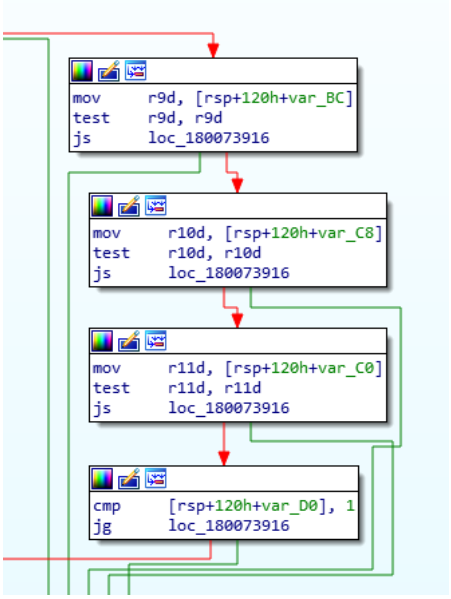
Obrázek 5.6: Testování konce souboru ze zranitelnosti cve_2021_29363.

CVE-2021-29361 Stejně jako u většiny zranitelností a oprav tohoto programu není jednoduché tuto zranitelnost přesně nalézt. Zranitelnost byla publikována 28. 9. 2021 a měla by umožňovat spuštění kódu neboli buffer overflow (přetečení bufferu). Zranitelnost není příliš dobře zdokumentovaná, ale i tak se mi podařilo nalézt nejpravděpodobnější místo opravy této zranitelnosti a důvod, proč tato zranitelnost vzniká. Při prozkoumání souboru vyvolávající toto chování jsem si všiml, že operační systém Windows nedokáže rozeznat metadata tohoto souboru, což jsem sám mohl pozorovat při upravování různých ostatních souborů, kupříkladu bmp. Windows tedy pravděpodobně při čtení hlavičky souboru kontroluje přímo velikost z hlavičky proti reálné velikosti souboru. Z tohoto důvodu jsou kontrolována metadata ještě před reálnou alokací a čtením. Stejné chování je možné pozorovat u čtení souborů typu JPEG, kde se nejdříve načte několik bytů, ty se otestují a teprve následně, pokud je vše v pořádku, načte se zbytek.

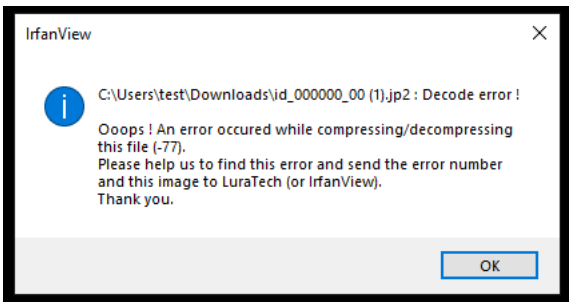
Na obrázku 5.7 je vidět pravděpodobné místo opravy této zranitelnosti. V originální zprávě o chybě se chyba vyskytuje až o mnoho instrukcí dále, ale vzhledem k tomu, že se jedná o chybu při načítání a kontrole hlavičky, je vhodné kontrolu vložit již za funkci načítající hlavičku.

CVE-2021-29360 Stejně jako předchozí zranitelnost je tato zranitelnost z 28. 9. 2021 způsobena špatně validovanou hlavičkou RLE souboru. K tomuto závěru lze dojít již při pohledu na testovací soubory. Navíc jsou obě zranitelnosti opraveny stejným blokem kódu. Na předchozím obrázku 5.7 je možné vidět instrukci skoku JLE, která následuje po intrukci TEST **EAX**, **EAX**. Toto se může zdát jako špatné instrukce, které by neměly mít žádný efekt, ale ve skutečnosti instrukce JLE (Jump less equal) je závislá na několika příznacích. Tato instrukce skočí, pokud je **ZF** (Zero flag) nula, nebo pokud se **SF** (Signed flag) nerovná **OF** (Overflow flag). U této zranitelnosti se jedná o případ nuly na **ZF**.

CVE-2020-23565 Opět velmi špatně ověřitelná zranitelnost z 5. 11. 2021 umožňující možné spuštění kódu útočníkem. Jedná se o knihovnu JPEG2000.dll



Obrázek 5.7: Testování validní hlavičky.



Obrázek 5.8: Chybové hlášení u chyby cve_2020_23565.

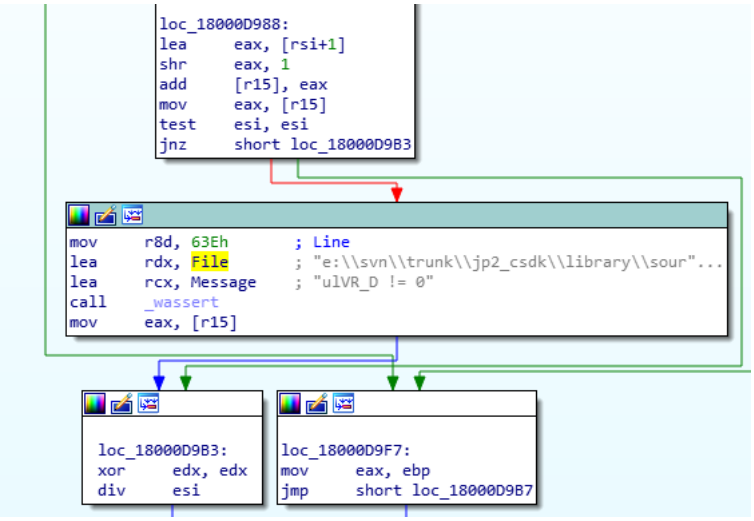
ve verzi 4.53, kterou se mi bohužel nepodařilo získat. Při prohledání dissassemblovaného programu se mi podařilo najít pravděpodobné místo zranitelnosti, o kterém autor píše. Jak lze předpokládat, je toto místo značně posunuto oproti předpokládanému místu díky rozptylu obou verzí knihovny. Vlastní chyba je opravena o několik úrovní výše tak, že se program ani do tohoto místa nedostane. Lze tedy předpokládat, že zde původně chyběla některá z podmínek. Bohužel s aktuální verzí nemohu zjistit, která z těchto podmínek způsobila chybu. Program IrfanView tedy snadno detekuje chybu při dekódování obrázku a zobrazí chybovou zprávu.

5.1.3 Odepření služby

Méně závažné zranitelnosti typu odepření služby jsou většinou častější a ani program IrfanView není výjimkou. Mohou se vyskytovat v několika formách,

jako je například nekonečné smyčky, vyčerpání paměti nebo permanentní uzamykání souborů.

CVE-2021-29365 Zranitelnost z 28. 9. 2021 využívající chybu v knihovně Effects.dll. Dle specifikace by se mělo jednat o vadnou funkci `AutoCrop_W`, která umožňovala vytvoření denial of service útoku na bázi nekonečné smyčky. Vlastní report nemá určené specifické místo, ale podařilo se mi nalézt vstupní soubor, který by měl toto volání evokovat. Po použití nástroje AutoCrop se program nikde nezacyklil. Navíc je tato knihovna ve verzi 4.59.10.0, což je vyšší verze než ta, která měla obsahovat danou chybu. Jmenovitě se jednalo o verzi 4.57.0.0. Po prozkoumání této funkce je možné říci, že míst, kde by se mohl program zacyklit, je hned několik, ale všechna jsou správně ošetřena.

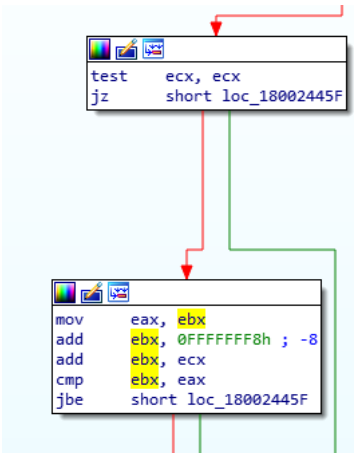


Obrázek 5.9: Ukázka opravy programu pomocí funkce wassert.

CVE-2020-23567 Tuto zranitelnost publikovanou 5. 11. 2021 se mi nepodařilo identifikovat, neboť se tato chyba vyskytovala v externí knihovně JPEG2000.dll ve verzi 4.53. Bohužel se mi nepodařilo získat původní verzi této knihovny, ale podařilo se mi objevit pravděpodobné místo původní zranitelnosti a její opravy. Jedná se o velice jednoduchou opravu pomocí funkce `_wassert`, která vytvoří výjimku, pokud by měl program pokračovat s hodnotou 0 v registru **ESI**.

CVE-2020-23566 Tato zranitelnost objevená 5. 11. 2021 má podobný problém jako zranitelnost CVE-2020-23567. Hlavní program lze v mezích možností získat v různých verzích, ale ostatní součásti programu, jako jsou například knihovny, už je velmi těžké získat ve starých verzích, a tak bude tato sekce spíše spekulací. V místě blízkém předpokládanému výskytu této chyby,

kterou je možné popsat jako denial of service pomocí nekonečné smyčky se ve verzi 4.56 nikde nic podobného nevyskytuje. Vzhledem k tomu, že se jedná o sekci programu velmi blízkou začátku segmentu .text, je možné předpokládat, že byla tato část při kompilaci přesunuta, nebo dokonce zcela přepsána. Vzhledem k provedené dynamické analýze se vstupním souborem z reportu o zranitelnosti se přikláním k přepsání celé této části autorem, neboť při krokování průchodu celou knihovnou jsem nenalezl žádnou část přesně odpovídající této zranitelnosti. Tato analýza je tedy spíše spekulací vycházející z chování programu při zobrazení obrázku demostrujícího toto chování.



Obrázek 5.10: Detekce pokusu o zacyklení při iterovaní sekcemi.

Jak je na obrázku 5.10 vidět, zde jsem určil nejpravděpodobnější místo původní zranitelnosti. Jedná se o cyklus určující, zdali byly načteny všechny sektory obrázku formátu JPEG2000. Pravděpodobně zde buď chyběla instrukce JBE short loc_18002445F, nebo nebyla správně napsána a předpokládala správný formát dat. Dle popisu zranitelnosti je klidně možné předpokládat, že tato kontrola špatného počtu segmentů měla původně pouze operátor rovná se. Pokud by toto byla pravda, bylo by možné předpokládat popisované chování. Program by měl vždy informaci, že se v souboru vyskytuje ještě jeden segment, který by se pokusil načíst/rozkódovat, ale tento segment by zde již nebyl.

CVE-2020-23549 Zranitelnost z 28. 10. 2021 umožňující denial of service útok na knihovnu Formats.dll. Bohužel se jedná o externí knihovnu, ke které se mi nepodařilo nalézt dřívější verze a hledání této zranitelnosti a její případné opravy je velmi složité. Celé záležitosti ani nepomáhá, že chyba byla nahlášena ve 32bitové verzi knihovny. Údajně by se mělo jednat o provedení či neprovedení určité větve programu na základě zfalšovaných dat. Ve zprávě o zranitelnosti je možné se dozvědět, že se jedná o denial of service společně s možným přetečením. Nepodařilo se mi nalézt místo v programu, kde se v minulosti

vyskytovala chyba. Dokonce jsem ani nenalezl v okruhu 10 KiB instrukci popsanou v této zprávě o zranitelnosti. Je tedy možné předpokládat, že byla celá tato část zcela přepsána.

Navíc jsem se pokusil odkrokovat program pomocí debuggeru, ale ani tak se mi nepodařilo najít místo odpovídající publikované zranitelnosti. Tato zranitelnost je navíc složitá k nalezení z důvodu, že autor, který ji objevil, nikdy nepublikoval ukázkový soubor demonstrující toto chování.

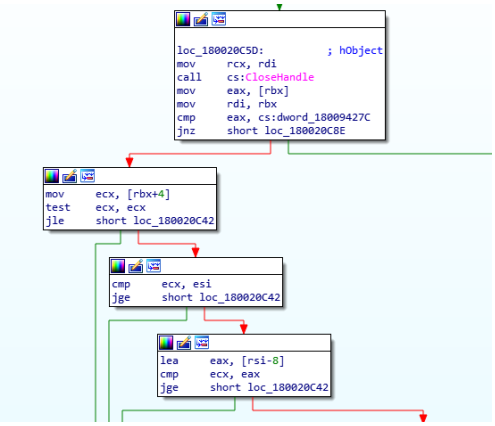
CVE-2020-23546 Stejně jako předchozí zranitelnost se mi tuto zranitelcost, publikovanou ve stejný den, tedy 28.10.2021 nepodařilo odhalit. Důvody nemožnosti nalézt zranitelnost popřípadě její opravu plynou ze stejných důvodů jako u zranitelnosti předchozí. Navíc je vhodné zmínit, že zranitelnost CVE-2020-23546 a CVE-2020-23549 objevili stejní lidé z NCSC Vietnam, tj. Vietnamský úřad pro národní kyberneticou bezpečnost.

5.1.4 Out of bounds

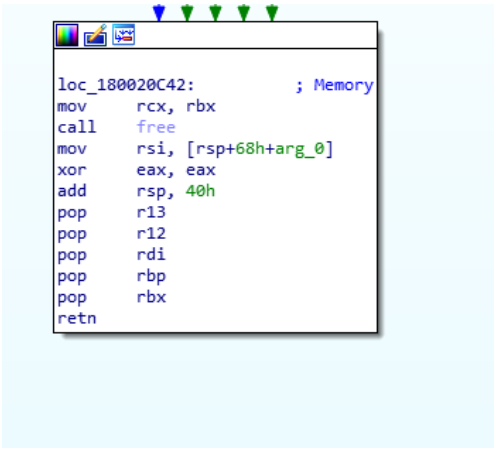
Tato sekce se zaměřuje na chyby typu out of bounds. Jedná se o zranitelnosti, kdy program čte před nebo za místem, které bylo pro tento účel vytvořeno.

CVE-2021-29358 Tato zranitelnost z 28. 9. 2021 využívá vlastnosti programu IrfanView, kde IrfanView většinou nezkoumá velikost souboru a pouze použije velikost, která je napsána v metadatech souboru. Druhý přístup, se kterým se setkáváme u programu IrfanView je ten, u kterého se naopak ignoruje velikost souboru z metadat, velikost souboru si program sám vypočítá kupříkladu pomocí funkcí `fseek` a `ftell`. V případě souboru typu PCX ale autor tyto dva přístupy zkombinoval dohromady. Výsledkem bylo, že program vytvořil paměť pro uložení načítaného souboru pomocí programem předpočítané velikosti. K tomuto úkonu byla využita funkce `FileSize`. Následně ovšem autor načítal soubor o velikosti, kterou určovala metadata. Z tohoto důvodu bylo velmi snadné přečíst data mimo specifikovanou oblast. Autor tuto zranitelnost opravil jednoduchým porovnáním velikostí souboru s velikostí z metadat. Pokud se tato dvě čísla nerovnají, je funkce ukončena s chybovou zprávou.

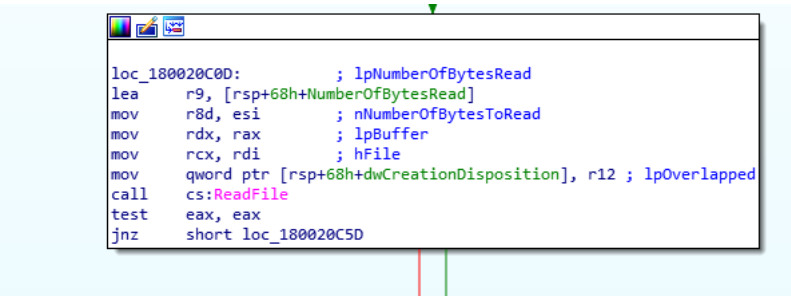
CVE-2020-35133 Zranitelnost z 16. 12. 2020, která umožňuje přepsat část paměti. Nedařilo se mi nalézt místo, kde by se tato chyba měla nacházat, neboť infomace podané nálezcem této zranitelnosti nekorespondovaly s informacemi, které jsem nalezl. Z tohoto důvodu jsem si sehnal IrfanView verzi 4.57 a podíval se přímo na verzi programu, ve které by se měla tato zranitelnost vyskytovat. Nakonec se mi podařilo zjistit, že se tato chyba vyskytovala pouze v 32bitové verzi prohlížeče obrázků IrfanView. Z tohoto důvodu není tato chyba pro tuto analýzu relevantní. I tak ale pro úplnost tuto sekci přikládám včetně Github stránky obsahující soubory vyvolávající toto chování [25].



(a) Porovnání naměřených velikostí s metadaty.



(b) Ukončení při detekci manipulace s velikostí v metadatech.



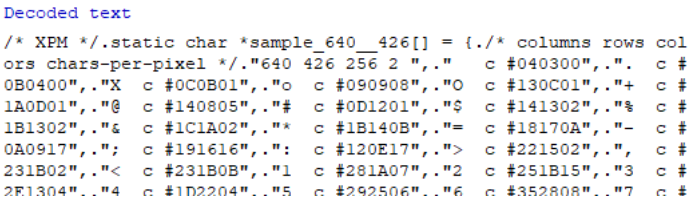
(c) Kontrola načtení nenulového počtu bytů.

5.1.5 Write access violation

Tato zranitelnost vzniká při neoprávněném přístupu k některé části paměti. Asi nejznámější je pro běžného programátora zápis na nulovou adresu. Tato adresa je samozřejmě write protected a vyvolá tedy výjimku. Nejedná se ovšem pouze o zápis na nulovou adresu, ale jedná se o jakoukoliv adresu do každého segmentu, který nemá příznak zápisu.

CVE-2020-23545 Zranitelnost publikovaná 15. 12. 2021 ukazující na zápis dat do nepovolené oblasti. Stejně jako u ostatních zranitelností nalezených vietnamskou NCSC je velmi obtížné nalézt vlastní zranitelnost nebo její opravu. U této zranitelnosti je ovšem vhodné pochopit, jakým způsobem vypadá formát souboru xpm neboli X PixMap. Každý, kdo zná programovací jazyk C, při zobrazení dat okamžitě pozná, že se jedná prakticky o volání funkce ve formátu jazyka C 5.12. Respektive pro úplnou přesnost je nutné říci, že se ani tak nejedná o funkci a její volání, ale spíš o definici 2D pole. Je možné předpokládat, že tento formát byl vytvořen z důvodu jednoduché integrace do zdrojového kódu. Tento přístup má ovšem i negativní vlastnosti. Jmenovitě se jedná o

možnost vkládání komentářů do různých částí souboru. Programy zobrazující obrázky v tomto formátu poté musí tyto komentáře rozeznat a při načítání vynechat a efektivně tak vytvořit drobnou část kompilátoru.



Obrázek 5.12: Obsah souboru ve formátu XPM.

V popisu zranitelnosti je možné se dočíst, že se toto chování týká instrukce mov byte ptr [ecx+ebp], al. Při bližším prozkoumání lze odhadnout, že tato instrukce se většinou vyskytuje u souborů s různou kompresí nebo u zpracovávání polí dat. Zde je myšleno kupříkladu zvýšení každého pixelu o nějakou hodnotu a podobné.

Při detailním prozkoumání celé komponenty se mi nepodařilo nikde objevit nic, co by byť vzdáleně připomínalo tuto instrukci nebo její chování. Vzhledem k tomu, že se v celé části knihovny zaobírající se načítáním souboru ve formátu X PixMap nepoužívají běžné instrukce pro čtení dat ze souboru, jako jsou například `fread`, je možné předpokládat, že vývojář po zjištění těchto zranitelností celou tuto funkci přepsal. V této funkci se pro načtení dat ze souboru používají pouze funkce `fgetc`, které jsou následně vždy kontrolovány pomocí funkce `feof`. Z těchto důvodů se domnívám, že autor celou funkci přepsal, aby odstranil všechny chyby.

CVE-2020-13906 Zranitelnost publikovaná 10. 6. 2020 na verzi IrfanView 4.54. Stejně jako předchozí se jedná o zranitelnost nalezenou uživatelem nhiephon z vietnamské NCSC. Opět se jedná o zranitelnost ve 32bitové verzi knihovny formats a je tedy velmi složité nalézt zranitelnost nebo její opravu. Jediné, co je ve zprávě napsáno, je, že tato write access violation vzniká při otevírání souboru typu HDR neboli High Dynamic Range souboru. Bohužel se mi zde nepodařilo nalézt místo opravy této zranitelnosti.

CVE-2020-13905 Jedná se o zranitelnost velmi podobnou předchozím zranitelnostem stejného typu. Tato zranitelnost byla nalezena 10. 6. 2020 ve verzi 4.54 32bitové verzi programu IrfanView. Zranitelnost byla nalezena uživateli Nguyn Quang a LMT z vietnamské NCSC. Většinu zranitelností v externích knihovnách je velmi složité nalézt primárně z důvodu, že nemám přístup k původním verzím těchto knihoven, a také z důvodu velkých změn v programu mezi různými verzemi. Navíc tato zranitelnost má svůj referenční bod (funkci `GetPlugInInfo`) velmi daleko (+0x38ed4), což velmi ztěžuje hledání této zranitelnosti. Po prohledání části programu, která má na starosti otevírání sou-

borů typu HDR se mi nepodařilo identifikovat přesné místo, kde by se mohla tato zranitelnost vyskytovat, a to ani při prozkoumání 32bitové verze této knihovny. Navíc, což je u většiny zranitelností programu IrfanView, autor programu v patch notes zmínil, že tuto zranitelnost opravil a nálezce této zranitelnosti s ním souhlasí.

5.1.6 Zranitelnosti a jejich opravy

Vývojář programu IrfanView Irfan Škiljan je velmi aktivní při opravách tohoto programu a zdá se, že většinu zranitelností opravuje ve velmi krátké době. Toto koresponduje s úvodem této práce, kde jsem zmínil, že se jedná o stále používaný a stále vyvíjený program. O rychlosti nasazení oprav jsem se mohl sám přesvědčit, neboť jsem v průběhu psaní této práce měl možnost pozorovat nahlášení nové chyby a její následné opravení v následující verzi programu. Jedná se o chybu CVE-2021-46064 vyskytující se ve verzi 4.59. Ve 32bitové verzi programu bylo možné vytvořit přetečení při otevření souboru typu TIFF.

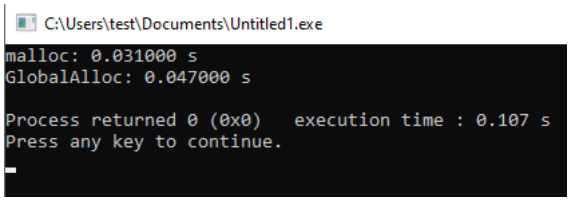
5.2 Nalezené nedostatky

Asi jako v každém jiném programu, tak se i v programu IrfanView vyskytují nedostatky. Některé jsou více závažné a jiné spíše kosmetické. Z tohoto důvodu jsem se rozhodl je rozřadit do několika skupin. Je velmi důležité říci, že některé skupiny nedostatků nejsou dobře ospravedlnitelné, neboť nemusí být chybou vývojáře. Primárně tím myslím neefektivní strojový kód ve smyslu opětovného výpočtu nebo opakování kódu. Tyto nedostatky jde z hlediska vývojáře odstranit, ale většinou vyžadují velmi důkladný přístup při psaní zdrojového kódu. Je tedy možné za většinu malých chyb obvinit jak vývojáře, tak kompilátor, který při optimalizaci nedokázal prohlédnout některé konstrukty. Pravdou ovšem také zůstává, že některé z těchto nedostatků vycházejí z volby použitého jazyka.

5.2.1 Čtení neinicializované paměti

Různé části programu IrfanView používají různé techniky práce s pamětí. Velmi často je kupříkladu použita funkce `GlobalAlloc`, které alokuje určitý počet byte a navíc přijímá argument specifikující její chování. Tyto argumenty jsou označovány jako Flagy a nabývají různých hodnot, jako jsou GHND, GMEM_FIXED, GMEM_MOVEABLE, GMEM_ZEROINIT, GPTR. GMEM_MOVEABLE, s hodnotou 0x0002 určuje možnost přesunu paměti. Takto naalokovaná paměť nemůže samozřejmě být přesunuta ve fyzické paměti, ale pouze její adresa v rámci haldy. Pro přístup k takovéto paměti je nutné použít funkci GlobalLock, která přemění HANDLE na ukazatel na specifickou paměť. Oproti tomu GMEM_FIXED slouží k alokaci nepřesouvatelné paměti. Tato varianta funkce má řídící flag 0x0000 a je použita, pokud není

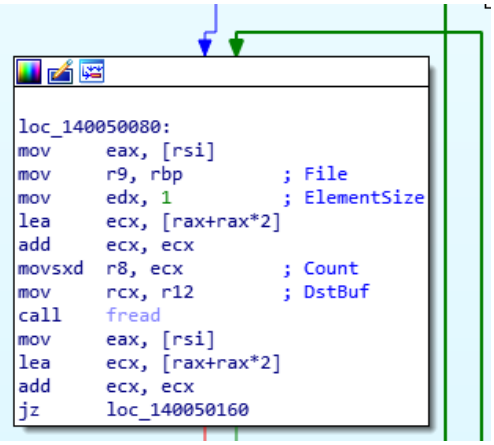
specifikována žádná jiná. Velmi důležitá je hodnota GMEM_ZEROINIT s hodnotou 0x0040, díky které se `GlobalAlloc` chová podobně jako funkce alloc. Zde je to myšleno v tom smyslu, že nuluje naalokovanou paměť. Zbylé hodnoty jsou GPTR s hodnotou 0x0040 pro kombinaci GMEM_FIXED a GMEM_ZEROPOINT, tedy nepohyblivá nulovaná paměť, a GHND (0x0042) pro pohyblivou nulovanou paměť. Funkce `GlobalAlloc` je téměř vždy v programu IrfanView použita pro alokaci a následné nulování nějaké části paměti. V několika případech, kdy je tato funkce použita bez GMEM_ZEROINIT, není možné získat informace z neinicializované paměti, neboť jsou tyto případy použity většinou pro alokaci tabulky adres členských funkcí nějaké třídy, tzn v constructoru. `GlobalAlloc` ale není jedinou alokační funkcí vyskytující se v programu IrfanView. Ve vlastním programu jsou navíc použité klasické funkce jako `malloc` a `calloc`. Jak bylo psáno dříve, funkce `calloc` je svým chováním velmi podobná funkci `GlobalAlloc` a flagem pro nulování naalokované paměti. Není tedy v hlediska hledání chyb programu příliš zajímavá. Funkce `malloc` ovšem může být potencionálně velmi nebezpečná, pokud je použita špatně. Na rozdíl od ostatních je ale velmi rychlá 5.13 a nepotřebuje příliš mnoho zdrojů, jako je například procesorový čas. Pro pochopení, proč je tato funkce zajímavá, je nutné zjistit, jak vůbec pracuje. Při jakékoliv alokaci je nutná spolupráce operačního systému s programem požadujícím nějakou paměť. První, co se stane, je získání nějakého kusu fyzické paměti operačním systémem. Tato paměť má svou fyzickou adresu, ale ta není pro program příliš vhodná, protože většinou programy mají potřebu iluze, že běží na systému samy. Z tohoto důvodu přeloží operační systém tuto fyzickou adresu na virtuální. K tomu slouží různé překladové tabulky, ale detaily nejsou v tomto případě důležité. Následně je operačním systémem předána virtuální adresa programu, který s ní již pracuje, jak se mu zlíbí. Je důležité poznamenat, že operační systém zde stále dělá jakéhosi prostředníka mezi fyzickou a touto virtuální pamětí [30].



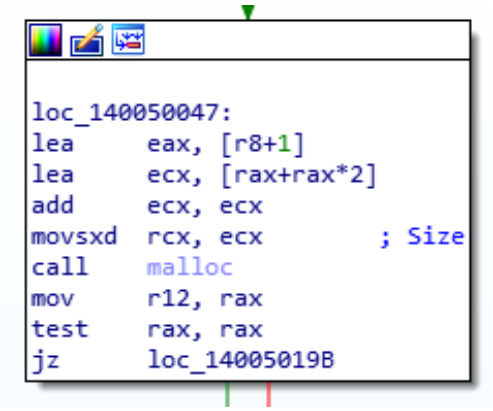
Obrázek 5.13: Rozdíl rychlostí dvou alokačních funkcí použitých v programu.

Je vhodné zmínit, že pokaždé, když program uvolní alokovanou paměť, nemusí být okamžitě paměť dealokována. V určitých případech je výhodné ponechat paměť v jakémsi poloalokovaném stavu, protože pokud si program vyžádá znovu stejně velkou paměť, je vyhodnější mu dát stejnou, kterou v minulosti vlastnil.

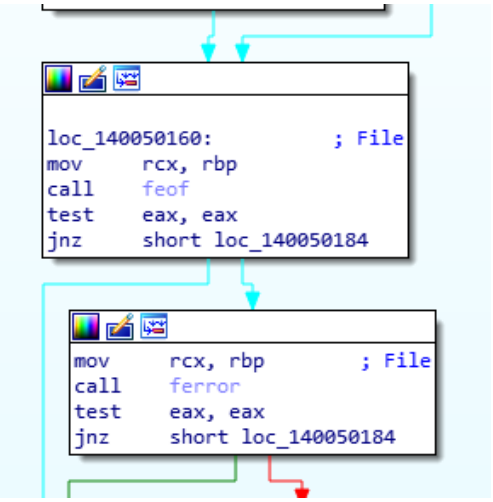
Pokud jsou při alokaci použity funkce nulující výslednou paměť, není příliš mnoho problémů. Jediná nevýhoda, jak bylo již zmíněno, je větší časová



Obrázek 5.15: Čtení souboru dle metadat.



Obrázek 5.14: Neinicializovaná alokace.



Obrázek 5.16: Ukončení načítání bez detekce počtu načtených znaků.

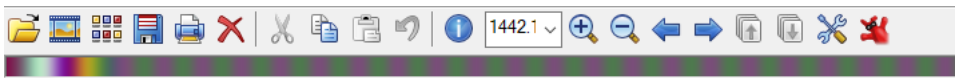
režie. Pokud si ovšem vývojář programu může být jistý, že bude daná paměť přepsána, není toto chování žádoucí, protože pouze zpomaluje program. V takovémto případě je lepší použít funkci `malloc`, která s alokovanou pamětí nijak neinteraguje a rovnou ji předává programu, který si ji vyžádal. Toto ovšem může představovat značné bezpečnostní problémy. V nejhorším případě je možné získat data z nějakého jiného programu, který svou paměť v minulosti dealokoval. Druhá, méně závažná možnost je získání dat z aktuálního programu, čímž je možné získat různé klíče a citlivá data. Ani jeden z těchto případů není zádoucí a měl by se jej vývojář co nejlépe vyvarovat [31]. Tyto zranitelnosti jsou spíše teoretické, neboť při allokaci paměti je starostí operačního systému znehodnotit paměť vrácenou jiným procesem. O toto znehodnocení

se stará takzvané nulující vlákno, které by mělo být přítomno v jakémkoliv operačním systému s certifikací pro práci s tajnými informacemi. Programovací jazyky jako je C++ a Java automaticky inicializují/znehodnocují získanou paměť. Jazyk C ovšem toto nedělá, a tak je tato zranitelnost závislá primárně na operačním systému a jeho bezpečnostní politice alokování paměti [32].

Ve vlastním programu IrfanView je možné nalézt několik použití funkce `malloc`, které jsou správně ošetřené. Ovšem pokud budeme požadovat zobrazení obrázku ve formátu NetPBM, je možné číst neinicializovanou paměť. NetPBM je jednoduchý formát uchovávání bitmapového obrázku, který má všechna metadata v ascii formátu. Vlastní soubor lze rozdělit na hlavičku a tělo, kde tělo obsahuje vlastní data a hlavička určuje typ a velikost obrázku. V hlavičce obrázku jsou tři velmi důležité informace: šířka, výška a maximální hodnota každého pixelu. Tento formát umožňuje použití různých maximálních hodnot včetně standardních 255. Pokud přinutíme program, aby předpokládal, že výsledný pixel je kódován ve více než jednom bytu, alokuje IrfanView paměť vycházející z jednoduché rovnice: výška*šířka*bajtů_na_pixel. Jak je vidět na obrázku 5.14 tato hodnota je následně alokována pomocí funkce `malloc` [33].

Navíc je vidět, že po alokaci a následném přečtení dat ze souboru se nekontroluje, zdali byla přečtena požadovaná velikost nebo se nachází program na konci programu. Irfanview dále pokračuje, jako kdyby přečetl validní soubor s tímto formátem a pokusí se ho zobrazit. Tímto chováním ovšem zobrazí neinicializovanou paměť, což je chybou.

Zde je třeba trochu polemizovat, jak moc velký problém je toto chování. Bohužel je toto už nezdokumentované teritorium a není tedy možné s jistotou říci, že pomocí této chyby lze číst cizí paměť. Vše totiž vychází z operačního systému a aktuální sestavy. Vlastní testování tohoto chování bylo v této práci uděláno ve virtualizovaném operačním systému, a není tedy jisté, zdali není požadovaná paměť předzpracována před předáním virtuálnímu systému. Při testování se vždy v nově naalokované paměti vyskytovaly hodnoty 0x0D, 0xF0, 0xAD, a 0xBA. Nepodařilo se mi dohledat, zdali tyto hodnoty mají něco společného s operačním systémem nebo spíše virtualizačním programem.



Obrázek 5.17: Výsledek čtení neinicializované paměti.

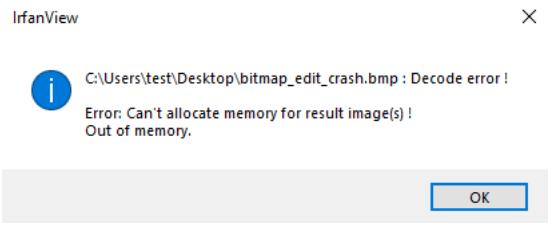
Oficiální metrikou CVSSv2 pro tuto nalezenou zranitelnost jsou následující hodnoty: **(AV:L/AC:L/Au:N/C:P/I:N/A:N)**

Tedy celkové skóre: **2.1**

5.2.2 Možné útoky odepření služby

Útoky typu odepření služby jsou většinou nejjednodušší útoky, které je možné použít proti různým cílům. Ani program IrfanView není výjimkou a bylo

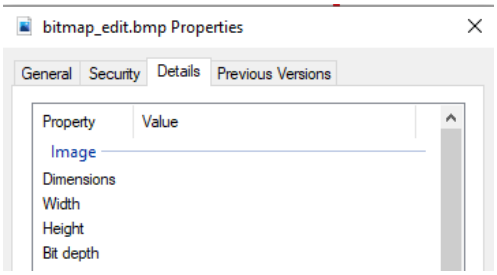
poměrně pravděpodobné, že se zde budou vyskytovat různé zranitelnosti umožňující tento typ útoku. Jak bylo psáno v teoretické části, tento útok většinou spočívá ve spotřebě výpočetního času daného systému. V případě obrázkového editoru je toto samozřejmě také možné, ale navíc je zde velké riziko denial of service útoku, který cílí přímo na paměť. Toto je i případ programu IrfanView. Většina různých formátů obrázků má nějaký způsob, jak předat informaci o velikosti daného obrázku. Může se jednat o rozlišení a barevnou hloubku, velikost celého souboru a jiné. Kupříkladu u jednoduchého formátu bitmap (bmp) se v určitých verzích vyskytuje jak rozlišení (barevná hloubka), tak i velikost těla celého obrázku. Navíc se v hlavičce tohoto souboru vyskytuje ještě jedna informace a sice, na jaké pozici začíná tělo celého souboru.



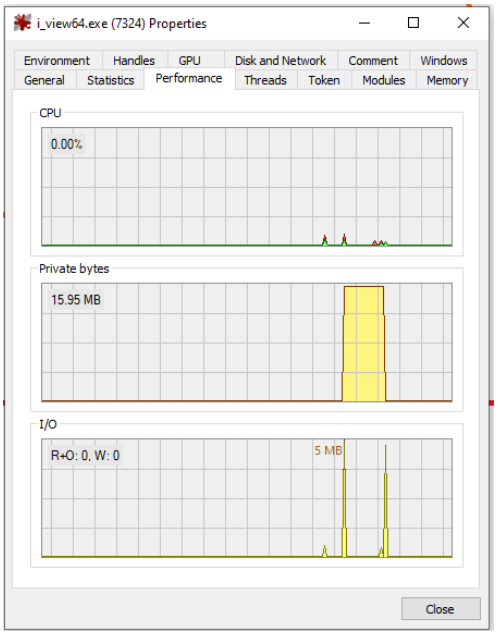
Obrázek 5.18: Chyba při pokusu o alokaci moc velkého bloku paměti.

Jak je vidět na obrázku 5.18, program skončí s chybou, pokud je formát obrázku natolik poškozen, že obsahuje neplatnou strukturu. Tyto kontroly jsou po většinu času velmi svědomitě dodržovány a není tedy jednoduché vytvořit vstupní obrázek, který bude poškozen, ale přesto se načte. V několika místech se ovšem i tak vyskytují různé chyby, které dovolují průchod poškozeného obrázku. Jak bylo řečeno dříve, právě u formátu bitmap se může vyskytovat informace o velikosti celého souboru a jeho rozlišení. Hodnota velikosti celého souboru se v tomto případě načte, ale nijak se s ní dále nepracuje. Následně se načte požadované rozlišení a naalokuje se podle této informace paměť. V průběhu četby zbytku hlavičky se vždy kontrolují informace správně a každá odchylka od standardu ukončí načítání a varuje uživatele. Bohužel v případě načítání těla obrázku tato kontrola chybí. Pokud je tedy velikost souboru pouze nesmyslné číslo, programu IrfanView to nevadí a zobrazí obrázek, jako by se nic nedělo.

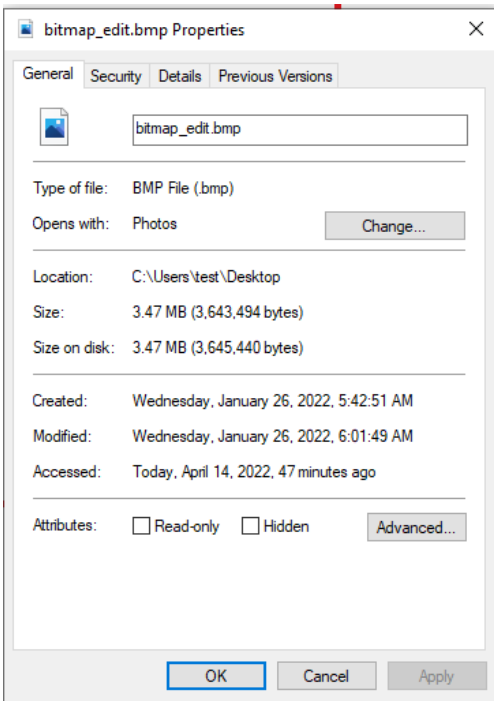
Navíc se nekontroluje načtení celého souboru. Program IrfanView se u některých formátů spokojí pouze s korektně načtenou hlavičkou. Je tedy možné zkonstruovat soubor ve formátu bitmap, který má ve skutečnosti velikost několik bytů, ale podle hlavičky by měl obsahovat velké množství dat. Právě z tohoto důvodu je velmi snadné vytvořit soubor, který se z pohledu operačního systému bude tvářit jako malý soubor, ale při načtení vyalokuje celou paměť. Naštěstí je v tomto případě při alokaci použita funkce `GlobaAlloc` s nulovacím příznakem, a není tedy možné snadno vyčítat paměť a zobrazovat ji.



Obrázek 5.19: Přečtená metadata operačním systémem.



Obrázek 5.20: Ukázka spotřebovaných zdrojů při načtení upraveného souboru.



Obrázek 5.21: Ukázka reálné velikosti souboru.

Oficiální metrikou CVSSv2 pro tuto nalezenou zranitelnost jsou následující hodnoty: **(AV:L/AC:L/Au:N/C:N/I:N/A:P)**

Tedy celkové skóre: **2.1**

5.2.3 Starý zdrojový kód

Při analýze programu IrfanView je velmi zřetelné, že buď byly velké části programu psány před mnoha lety, nebo pouze programátory, kteří rádi využívají zastaralé funkcionality. Je velice snadné nalézt spoustu příkladů volání funkcí, které jsou dle dokumentace zastaralé a neměly by se již používat. Toto varování ovšem není odůvodněné bezpečnostním rizikem, jako například funkce

`gets`. Tyto funkce, jako například GetDialogWindow, nejsou samy o sobě příliš nebezpečné, ale přímo dokumentace MSDN doporučuje použití novějších variant těchto funkcí. Hlavní důvod tohoto doporučení je to, že nové funkce mají většinou lepší vlastnosti nebo jsou lépe „tvarovatelné“. Strategie nechávání starých funkcí ve zdrojovém kódu je v tomto případě pochopitelná, protože vývojář nemusí vynakládat zdroje na rozsáhlé přepisování zdrojového kódu, který je stále funkční a bezproblémový. Navíc díky této strategii může vývojář zajistit kompatibilitu se staršími verzemi programu. Nejedná se ovšem o optimální strategii vzhledem k bezpečnosti/udržitelnosti programu. Jednoduchý příklad takového chování je u programů, které využívají spustitelného stacku. Takovýto program může fungovat na moderním systému, bude stále stejně bezpečný jako v době, kdy byl napsán, ale otázkou je, zdali je takovéto chování preferovatelné.

Další druh zastaralých volání jsou takzvané nebezpečné funkce. Většina základních funkcí v programovacím jazyce C/C++, zejména pak funkce manipulující s polem bytů, nemají obrany proti přetečení. Je tedy díky těmto funkcím možné způsobit nějaký druh útoku spočívající v přepsání paměti. Takovouto chybu se mi nepodařilo při analýze zdrojového kódu odhalit a je tedy možné, že se zde takováto chyba nevyskytuje. Při volání těchto „nebezpečných“ funkcí autor programu vždy důsledně kontroluje velikosti argumentů. Je velmi pravděpodobné, že tyto kontroly jsou prováděny z určité části manuálně, neboť jich je v některých případech příliš mnoho. Více viz kapitola Neoptimální kód. Mezi tyto potencionálně nebezpečné funkce patří například funkce `wcsncpy`, která by při špatném zacházení mohla číst data i mimo své argumenty a přepisovat je. Alternativou k této funkci je funkce `wcsncpy_s`, která mimo všechny běžné argumenty, které přebírá funkce původní, ještě vyžaduje vložení maximální velikosti cílové paměti díky argumentu numberOfElements. Tím pádem tato bezpečnější funkce nepřekoná hranice paměti, se kterou má povoleno pracovat, a nemůže tedy nic přepsat, pokud je maximální velikost korektně zadaná.

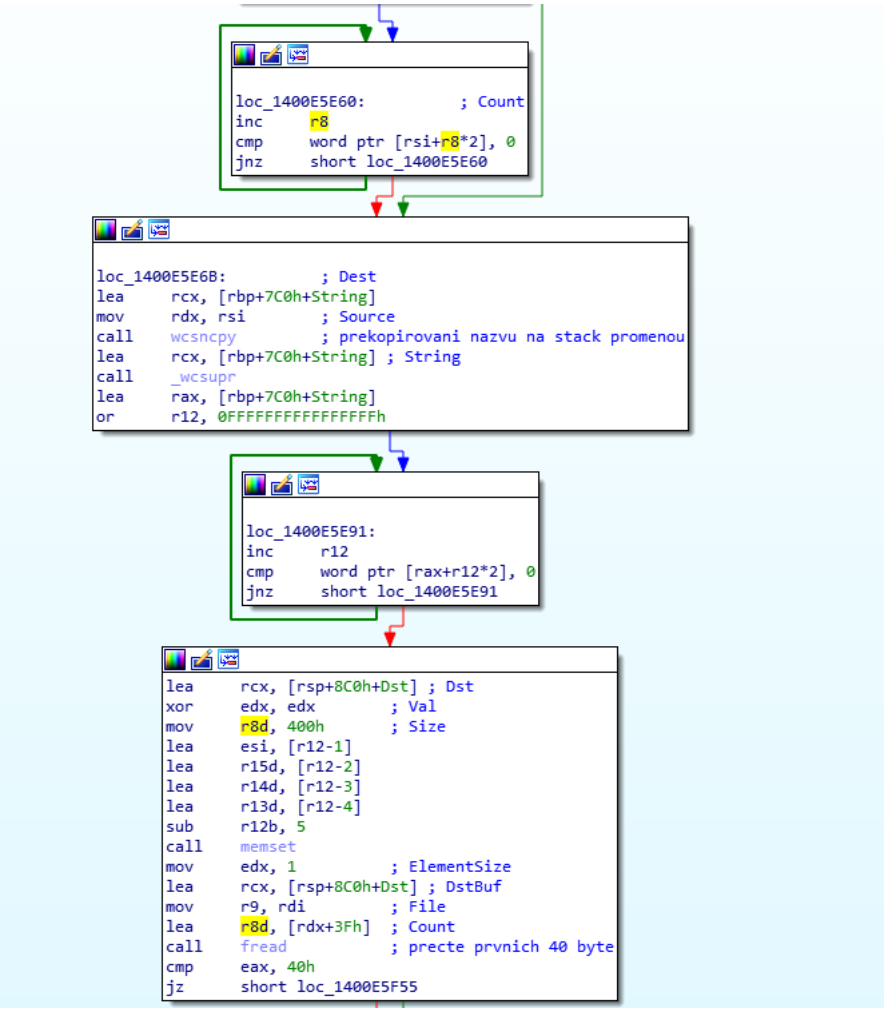
Nejedná se o zranitelnost a není tedy možné zařadit do CVSSv2 metriky.

5.2.4 Neoptimální kód

Při prvním pohledu se může zdát, že je zdrojový kód pouze čistý programovací jazyk C s některými C++ funkcemi pro vytvoření grafického rozhraní. Tento první pohled potvrzuje i výstup z programu Resource Hacker, který ukazuje, že celý zdrojový kód je kompilovaný kompilátorem MSVC++ 8.0. Při důkladném pohledu je ovšem zřejmé, že různé části tohoto programu byly vyvíjeny odlišně a mají tedy jinou strukturu. Pokud se zaměříme na některé obrázkové formáty, je možné odhalit poměrně přímočarou strukturu funkcí. Pokud se ovšem zaměříme kupříkladu na formát JPG, je možné odhalit sofistikovanou strukturu tříd s různým děděním. Je ovšem také možné, že autor

kupříkladu obrázek ve formátu JPG načítá pomocí objektově napsaného zdrojového kódu právě z důvodu komplexnosti celého formátu.

Kompilátor má ovšem i jiné nedostatky, které nejsou pouze takto kosmetické. Na mnoha místech je možné odhalit až příliš paranoidní kontrolu různých datových polí a struktur. Některé z těchto kontrol jsou zajisté nezbytné, ale v některých případech jsou pouze zbytečné a zpomalují běh celého programu. Tyto konstrukty byly pravděpodobně zavedeny do zdrojového kódu přímo autorem programu. Některé jsou velmi složité, a tak je pravděpodobně nedokáže kompilátor prohlédnout a optimalizací se jich zbavit. Většinou se ovšem nejedná o žádné velké prohřešky a jejich oprava není příliš nutná.



Obrázek 5.22: Neefektivní část programu provádějící opakovanou kontrolu velikosti řetězce.

Neoptimální kód se v programu projevuje i jinými způsoby. V kapitole zabývající se různými zdroji jsem zmiňoval kupříkladu různé řetězce znaků,

které program často používá. Ne vždy je ovšem práce s těmito zdroji efektivní. Zde je dokonce pravděpodobné, že se také jedná o nedostatek při optimalizaci zdrojového kódu, neboť v každém místě, kde se používají zdroje, je možné pozorovat toto chování. Většinou se jedná o nahrání zdrojů mnohem dříve, než je třeba. Zkusím toto chování připodobnit ke zjednodušenému příkladu volání funkce, která má dva možné průchody. V případě, že by se funkce chovala stejně jako u programu IrfanView, nahrávala by zdroje již na začátku těla funkce, i když je tento zdroj použit pouze v jedné větvi programu. Jedná se tedy o zbytečnou instrukci pro větev, která tento zdroj nepotřebuje. Samozřejmě je zde možné argumentovat tím, že načtení zdrojů z hlavní paměti trvá velmi dlouho, a je tedy vhodnější začít tato data načítat dříve. Toto je samozřejmě validní pohled na věc, pokud je pravděpodobnost průchodu jednou nebo druhou větví rovna. Pokud ovšem je jedna z větví mnohem pravděpodobnější, je třeba tento průchod upřednostnit zejména, pokud funkce mimo nahrávání tohoto zdroje pracuje i jinak s pamětí. Pokud se vrátíme k programu IrfanView, je zde možné vidět zbytečné načítání různých zdrojů, které většinou označují nějakou chybovou zprávu. Je pravděpodobné, že program na zobrazování a editaci obrázků nebude často využíván pro práci, jejíž výsledkem bude chybné načtení. Jedná se ovšem pouze o drobný nedostatek a z hlediska uživatele není možné pozorovat rozdíl. Ve výsledku se jedná o časový rozdíl v řádu nanosekund, protože takovéto načítání není nikde spojeno s načtením jiných částí paměti.

Stejně jako u předchozí sekce, ani zde se nejedná o zranitelnost a není tedy možné klasifikovat dle CVSSv2.

AltaLux.dll V pluginu Altalux.dll se mi nepodařilo nalézt žádné závažné nedostatky. Pouze se zde vyskytuje určité plýtvání pamětí. Proměnná FilterIntensity označuje intenzitu aplikovaného filtru, která je následně přepsána do řetězce. Ve skutečnosti je ale vytvořen příliš velký prostor pro znakovou reprezentaci tohoto čísla, což by mohlo způsobit složitější práci stránek paměti stacku. Jedná se ovšem pouze o drobnost.

```
531                                 char FilterString[256];
532                                 SkipProcessing = false;
533                                 sprintf(FilterString, "%d", FilterIntensity);
```

Obrázek 5.23: Sekce plýtvající pamětí v knihovně Altalux [14].

5.3 Zhodnocení zranitelností a oprav

Myslím si, že se mi podařilo analyzovat všechny typy zranitelností programu IrfanView a způsob, jakým na ně autor reagoval. Mohu tedy s jistotou říci, že se nejedná o zapomenutý program a že autor se jej snaží stále opravovat a

vylepšovat. K tomuto závěru jsem došel díky tomu, že se mi nepodařilo nalézt jedinou neopravenou zranitelnost, která byla v minulosti nahlášena.

Závěr

Cílem této práce byla bezpečnostní analýza programu na prohlížení obrázků IrfanView. Protože se jedná o poměrně starý, ale stále používaný program, bylo možné předpokládat, že některé části tohoto programu nemusejí být příliš udržované a opravované. Navíc vývojářem tohoto programu není velká firma, ale pouze jeden člověk Irfan Škiljan (a někteří vyvojáři pomocných knihoven).

Z tohoto důvodu jsem provedl extenzivní analýzu programu IrfanView, která měla za cíl zjistit, zdali nebyl program zanechán v neopraveném stavu. Při použití několika technik analýzy jsem došel k závěru, že je program velmi dobře udržován a nejedná se tedy o bezpečnostní riziko. Zranitelnosti, které jsem při analýze tohoto programu objevil, nejsou kriticky závažné a nejedná se tedy o velká rizika. Jmenovitě se jedná o zranitelnost umožňující útoky typu odepření služby (DoS) a čtení neinicializované paměti. Čtení neinicializované paměti může mít velký dopad na bezpečnost celého systému. Poté jsem tyto zranitelnosti ohodnotil pomocí systému CVSSv2 jako nekritické zranitelnosti.

Nakonec jsem zmapoval všechny druhy zranitelností programu IrfanView, které se v něm v minulosti vyskytly. Tyto zranitelnosti jsem následně detailně prostudoval a nalezl v disasseblovaném kódu programu jejich opravy. Tyto opravy jsem vyhodnotil jako velmi dobré, což svědčí o aktivitě vývojáře programu. Dokonce jsem byl svědkem jedné z těchto oprav, kde od nahlášení po vydání nové opravené verze uplynulo pouze několik dní.

Protože je program neustále vyvíjen, byla v průběhu vytváření této práce vydána nová verze programu a některých knihoven. Z tohoto důvodu doporučuji opětovnou analýzu nové verze programu IrfanView.

Literatura

[1] Skiljan, I.: What is IrfanView? `https://www.irfanview.com/main_what_is_engl.htm`, 2022.

[2] Gagnon, M. N.; Taylor, S.; Ghosh, A. K.: Software Protection through Anti-Debugging. *IEEE Security Privacy*, ročník 5, č. 3, 2007: s. 82–84, doi:10.1109/MSP.2007.71.

[3] Yan, W.; Zhang, Z.; Ansari, N.: Revealing packed malware. *ieee seCurity & PrivaCy*, ročník 6, č. 5, 2008: s. 65–69.

[4] Peres, M.: Reverse engineering power management on NVIDIA GPUs-A detailed overview. *Power*, ročník 75, č. 75W, 2013: str. 150W.

[5] Hirokawa, N.: *Automatic Stub Generation for Dynamic Symbolic Execution of ARM binary*. Dizertační práce, Japan Advanced Institute of Science and Technology, 2021.

[6] Saffaf, M. N.: Malware Analysis. 2009.

[7] Blem, E.; Menon, J.; Vijayaraghavan, T.; aj.: ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Transactions on Computer Systems (TOCS)*, ročník 33, č. 1, 2015: s. 1–34.

[8] Lomont, C.: Introduction to intel advanced vector extensions. *Intel white paper*, ročník 23, 2011.

[9] Brandejs, M.: *Mikroprocesory Intel*. Grada, 1991.

[10] Friedl, S.: Intel x86 JUMP quick reference. `http://unixwiz.net/techtips/x86-jumps.html`, 2022.

[11] Robertson, C.: Decorated Names. `https://docs.microsoft.com/en-us/cpp/build/reference/decorated-names?view=msvc-170`.

[12] Mell, P.; Scarfone, K.; Romanosky, S.; aj.: A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-forum of incident response and security teams*, ročník 1, 2007, str. 23.

[13] Lhee, K.-S.; Chapin, S. J.: Buffer overflow and format string overflow vulnerabilities. *Software: practice and experience*, ročník 33, č. 5, 2003: s. 423–460.

[14] StefanoT: Altalux-IrfanView. https://github.com/StefanoT/AltaLux-IrfanView/blob/f32eadd6a42635318a904f34789232f3ee124170/AltaLux/AltaLux.cpp, 2018.

[15] Eyrich, B.: IrfanView AWD Plugin. https://www.artweaver.de/en/help/159.

[16] Coralic, M.: BabaCAD. https://www.artweaver.de/en/help/159, 2012.

[17] ArchieMC: IrfanView. https://sourceportal.blogspot.com/2017/08/irfanview.html, 2017.

[18] Skiljan, I.: IrfanView Plugins. https://www.irfanview.com/plugins.htm.

[19] team charls: charls. https://github.com/team-charls/charls, 2022.

[20] ilyakurdyukov: jpeg-quantsmooth. https://github.com/ilyakurdyukov/jpeg-quantsmooth, 2022.

[21] Italia, M.: IrfanPaint. https://mitalia.net/irfanpaint/.

[22] Szekely, I.: IrfanView Region Capture Plugin. https://grebulon.com/software/irfanview_region_capture.php.

[23] lvm, B. P.: SVG plugin doesn't work very well. https://irfanview-forum.de/forum/program/support/91974-svg-plugin-doesn-t-work-very-well.

[24] moshekaplan: Research. https://github.com/moshekaplan/Research/tree/main/IrfanView, 2021.

[25] DmitryMeD: pentesting. https://github.com/DmitryMeD/pentesting/blob/main/IrfanView%204.56.md, 2021.

[26] KamasuOri: publicResearch. https://github.com/KamasuOri/publicResearch/tree/master/poc/irfanview/2, 2020.

[27] nhiephon: Research. https://github.com/nhiephon/Research/blob/master/README.md, 2022.

[28] Özkan, S.: Irfanview : Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list.php?vendor_id=317&product_id=0&version_id=0&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=0&cvssscoremax=0&year=0&cweid=0&order=1&trc=157&sha=703e8ccfafae14f9b2c75f71f5c4ff1a1f8d55aa.

[29] Ashcraft, A.: Maximum Path Length Limitation. https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd.

[30] Corporation, M.: GlobalAlloc function (winbase.h). https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-globalalloc, 2021.

[31] Corporation, M.: malloc. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/malloc?view=msvc-170, 2021.

[32] davidb: Reading physical memory frame previously owned by another process to read contents of its memory page. https://security.stackexchange.com/questions/110477/reading-physical-memory-frame-previously-owned-by-another-process-to-read-conten.

[33] pnm. http://netpbm.sourceforge.net/doc/pnm.html, 2013.

Příloha A

Seznam použitých zkratek

ARM Advanced RISC Machines

BP Base Pointer

CF Carry Flag

CISC Complex instruction set computer

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

DoS Denial of Service

GIF Graphics Interchange Format

IP Instruction Pointer

MMU Memory Management Unit

MSVC++ Microsoft Visual C++

NCSC National Cyber Security Center

OF Overflow Flag

PF Parity Flag

RISC Reduced instruction set computer

SF Sign Flag

SI Source Index

SP Stack Pointer

TIF Tagged Image File

ZF Zero Flag

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `practical`
    - `database` ..... databáze s projekty
    - `iv.exe` ..... instalační soubor programu IrfanView
    - `inputs` ..... vstupní soubory
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS