Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Obfuskačné techniky skupiny Stantinko

Bakalárska práca

Bratislava, 2021
Vladislav Hrčka

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Obfuskačné techniky skupiny Stantinko

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Aplikovaná informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Jaroslav Janáček, PhD
Konzultant:	Mgr. Peter Košinár

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Vladislav Hrčka
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický

Názov: Obfuskačné techniky skupiny Stantinko
Obfuscation Techniques of the Stantinko group

Anotácia: Stantinko je rodina malvéru aktívna aspoň od roku 2012, ktorá postupne vyvíjala a vylepšovala vlastné unikátne obfuskačné techniky, aby zkomplikovala prácu analytikom a vyhla sa detekcii. Práca sa zaoberá popisom týchto techník a možnosťami analýzy programov, ktoré nimi boli obfuskované.

Cieľ:
- analyzovať a popísať obfuskačné techniky používané skupinou Stantinko
- porovnať ich s podobnými a všeobecne známymi obfuskačnými technikami
- navrhnúť a implementovať riešenie uľahčujúce analýzy programov obfuskovaných týmito technikami

Vedúci:	RNDr. Jaroslav Janáček, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	21.10.2020

Dátum schválenia: 04.11.2020

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.... študent

.... vedúci práce

Poďakovanie

Rád by som sa poďakoval môjmu školiteľovi Jaroslavovi Janáčkovi za usmernenie a cenné pripomienky a konzultantovi Petrovi Košinárovi za odbornú pomoc a diskusie pri tvorbe práce.

Abstrakt

Stantinko je rodina malvéru, ktorá postupne vytvárala a vylepšovala vlastné obfuskačné techniky, aby sťažila analýzu a samotnú detekciu jej programov. Tento botnet pozostávajúci z viac ako pól milióna obetí je používaný jeho operátormi na rôzne kybernetické zločiny. V bakalárskej práci sa na začiatku zaoberáme opisom všetkých použitých obfuskačných techník. Tieto techniky následne porovnáme k už známym a podobným technikám, pričom prídeme na to, že vylepšenia, ktoré boli pridané k týmto inak bežným technikám, znemožňujú použitie už existujúcich nástrojov, ktoré by mali byť schopné zjednodušiť takto obfuskovaný kód. Nakoniec implementujeme a popíšeme riešenie, ktoré dokáže značne uľahčiť analýzu takto obfuskovaných programov.

Abstract

Stantinko is a malware family, which has been gradually improving its custom code obfuscation techniques to hinder analysis and detection. The half-million-strong Stantinko botnet has been used by its operators for various cybercriminal activities. In this Bachelor thesis we begin by analyzing the applied obfuscation techniques and subsequenly compare them to already known and similar techniques. It makes us realize that they introduced enhancements to some otherwise common techniques, which turn ordinary reverse engineering methods to deal with such techniques useless. At the end of the thesis, we implement and describe solution, which can facilitate analysis of code obfuscated in this way.

Obsah

Úvod....3
1 Východiská....4
1.1 Teoretické východiská....4
1.1.1 Teória grafov....4
1.1.2 Techniky analýzy kódu programov....5
1.1.3 Iné teoretické východiská....9
1.2 Technologické východiská....10
1.2.1 Miasm....10
1.2.2 IDAPython....17
1.2.3 RPyC....17
2 Analýza obfuskačných techník....18
2.1 Prvý kontakt....19
2.2 Obfuskácia toku riadenia....19
2.3 Mŕtvy kód....24
2.4 Irelevantný kód....24
2.5 Obfuskácia reťazcov....25
3 Popis podobných obfuskačných techník....26
3.1 Vyrovnávanie toku riadenia....26
3.2 Mŕtvy kód....28
3.3 Irelevantný kód....28
3.4 Stack strings....29
4 Deobfuskácia....30
4.1 Deobfuskácia vyrovnaného toku riadenia....30
4.1.1 Identifikácia vyrovnaného toku riadenia vo funkcii....31
4.1.2 Deobfuskácia identifikovaného vyrovnaného toku riadenia vo funkcii....43
4.1.3 Automatická deobfuskácia vyrovnávania toku riadenia v programe....46
4.2 Deobfuskácia reťazcov....49
5 Výsledky....50
6 Záver....59
Literatúra....60

Obrázky

Obr. 1: Ukážka disassemblovania kódu pomocou Miasm-u..12
Obr. 2: Ukážka assemblovania kódu pomocou Miasm-u...13
Obr. 3: Ukážka symbolického vykonávania pomocou Miasm-u...14
Obr. 4: Ukážka emulácie kódu pomocou Miasm-u...15
Obr. 5: Ukážka pripájania dynamického symbolického vykonávania k Sandbox-u v Miasm-e......15
Obr. 6: Ukážka inicializácie RPyC servera...18
Obr. 7: Ukážka pripojenia k RPyC serveru...18
Obr. 8: Ukážka bežného grafu toku riadenia obfuskovaných funkcií...19
Obr. 9: Ukážka štandardného obfuskačného konštruktu..20
Obr. 10: Ukážka obfuskačného konštruktu s prekryvom základných blokov (zvýraznené žltou)....22
Obr. 11: Ukážka predčasného návratu z obfuskačného konštruktu (v úseku 3)...............................23
Obr. 12: Ukážka obfuskačného konštruktu s viacerými chvostami (vľavo) a bez chvostov (vpravo).
....23
Obr. 13: Ukážka *irelevantného* kódu...25
Obr. 14: Ukážka skladania reťazca...26
Obr. 15: Ukážka vyrovnaného toku riadenia..27
Obr. 16: Ukážka mŕtveho kódu...28
Obr. 17: Ukážka irelevantného kódu...29
Obr. 18: Ukážka stack strings..30
Obr. 19: Ukážka závislých inštrukcií vetvenia od priradenia..35
Obr. 20: affected_lines vygenerované z predchádzajúcej ukážky..37
Obr. 21: Ilustrácia spätnej hrany pri vyrovnanom toku riadenia..39
Obr. 22: Ilustrácia okamžitých dominátorov vo vyrovnanom toku riadenia...................................40
Obr. 23: Ilustrácia deobfuskovanej funkcie..46
Obr. 24: Ilustrácia obfuskovanej funkcie..46
Obr. 25: Spustenie RPyC servera v IDE..51
Obr. 26: Náhľad obfuskovanej funkcie 1...52
Obr. 27: Volanie obfuskovanej funkcie 1...52
Obr. 28: Prázdna sekcia pre deobfuskovaný kód...53
Obr. 29: Kód pre deobfuskáciu funkcie 1...53
Obr. 30: Náhľad deobfuskovanej funkcie 1..54
Obr. 31: Ukážka deobfuskovania identifikovaných dosiahnuteľných a obfuskovaných funkcií......55
Obr. 32: Náhľad identifikovanej obfuskovanej funkcie 2..56
Obr. 33: Náhľad deobfuskovanej funkcie 2..57
Obr. 34: Náhľad funkcie s obfuskovaným reťazcom...58
Obr. 35: Kód, ktorý vyhľadá obfuskované reťazce..58
Obr. 36: Vyhľadané obfuskované reťazce..59

Úvod

Analýza malvéru je proces rekonštrukcie podozrivých súborov a ich porozumeniu. Vykonáva sa zvyčajne za účelom vyhodnotenia dopadu kybernetických útokov alebo nájdenia a extrahovania unikátnych prvkov či vlastností. Pomocou získaných údajov je možné identifikovať podobné súbory, ktoré boli videné v minulosti. Alebo ich, naopak, rozoznať, keď sa ukážu v budúcnosti. Na základe týchto indikátorov sa dá nakoniec napríklad odhaliť iné aktívne útoky, predísť budúcim alebo profilovať útočníka.

Metódy reverzného inžinierstva sú intenzívne využívané počas analýzy malvéru. V tomto prípade sa pod reverzným inžinierstvom myslí proces rekonštrukcie zdrojového kódu zo samotných spustiteľných súborov. Výsledný zdrojový kód sa má v čo najväčšej možnej miere podobať pôvodnému, ale vo väčšine prípadov nie je možné dosiahnuť úplnú zhodu, pretože sa pri kompilácii viacero informácii stráca.

Techniky, ktorých cieľom je spraviť reverzné inžinierstvo náročnejším zahmlievaním významu a komplikovaním výsledného kódu, sa nazývajú obfuskačné. Na ich zvrátenie sa vytvárajú deobfuskačné algoritmy, ktoré spravidla robia kód jednoduchším na pochopenie, pričom nemenia jeho význam. Obfuskačné techniky bývajú aplikované okrem legitímneho softvéru, pri ktorom slúžia primárne na ochranu duševného práva vlastníka, často aj na škodlivý, čím prirodzene komplikujú aj analýzu samotného malvéru.

Stantinko je rodina malvéru aktívna aspoň od roku 2012, ktorá postupne vyvíjala a vylepšovala vlastné unikátne obfuskačné techniky, aby zkomplikovala prácu analytikom a vyhla sa detekcii. Práca sa zaoberá popisom týchto techník a možnosťami analýzy programov, ktoré nimi boli obfuskované.

V prvej kapitole sa budeme zaoberať všetkými technickými a teoretickými východiskami práce. V druhej kapitole budeme analyzovať a popisovať obfuskačné techniky používané skupinou Stantinko. V tretej kapitole porovnáme dané techniky s podobnými a všeobecne známymi obfuskačnými technikami. V štvrtej kapitole opíšeme samotnú implementáciu výsledného kódu. V poslednej kapitole na niekoľkých ukážkach predvedieme, ako naše riešenie vyzerá v praxi.

1 Východiská

Nasledujúca kapitola obsahuje zhrnutie poznatkov z oblastí, ktoré sa týkajú tejto bakalárskej práce. Je rozdelená na dve hlavné časti.

Prvá časť sa zaoberá všetkými použitými poznatkami v teoretickej rovine a časť druhá použitými knižnicami.

1.1 Teoretické východiská

V tejto kapitole si priblížime teoretické východiská bakalárskej práce, ktoré sme rozložili na tri podkapitoly, kde:

- prvá vysvetľuje pojmy z teórie grafov;
- druhá sa venuje technikám analýzy kódu programov;
- tretia pokrýva ostatné teoretické východiská.

1.1.1 Teória grafov

V nasledujúcej kapitole objasníme význam menej známych pojmov z teórie grafov. Tieto pojmy budú prirodzene využívané neskôr v práci.

Strom. Strom je v teórii grafov taký graf, v ktorom sú akékoľvek dva vrcholy spojené práve jednou cestou.

DFS strom. DFS strom je strom, ktorý sa vytvoril pri prehľadávaní do hĺbky, pričom každý prvý krát navštívený vrchol bol spojený hranou s jeho predchodcom.

Spätná hrana. Máme daný ľubovoľný DFS strom grafu, spätná hrana je potom každá hrana, ktorá spája nejaký vrchol x s takým vrcholom, ktorý bol objavený skôr ako rodič vrcholu x. [11]

Dominátor. Vrchol *M* dominuje vrcholu *N* ak a iba ak všetky cesty z vybraného počiatočného vrcholu do vrcholu *N* vedú cez vrchol *M*. [10]

Striktný dominátor. Vrchol *M* striktne dominuje *N* ak a iba ak *M* dominuje *N* a *M* nie je *N*. [10]

Okamžitý dominátor. Vrchol *M* je okamžitým dominátorom *N* ak a iba ak *M* striktne dominuje *N* a zároveň každý ďalší striktný dominátor *N* dominuje *M*. [10]

1.1.2 Techniky analýzy kódu programov

Táto kapitola sa bude zaoberať vysvetlením pojmov určitých techník analýzy kódu programov. Kým začneme povieme ešte pár slov k tomu, čo tieto techniky vlastne sú.

Analýza kódu programov je proces automatického skúmania vlastností a správania programov. Techniky analýzy kódu programov sa delia na statické, ktoré sledovaný program nespúšťajú, a dynamické, pri ktorých je program analyzovaný za behu. My budeme používať najmä statické,

Intermediate representation. Intermediate representation (IR) je abstraktná reprezentácia programu obsahujúca určitú sadu inštrukcií, ktorá pripomína skôr assembler ako jazyky vyššej úrovne. Môže sa označovať aj ako medzikód. Takýto abstraktný jazyk by nemal byť závislý od konkrétnej platformy. [13, 14]

Použitie IR je výhodné z hľadiska znovu použiteľnosti kódu, teda v prípade, že sa má pridať nový jazyk alebo inštrukčná sada, nie je potrebné všetky algoritmy písať od začiatku, ale iba preložiť nový jazyk alebo inštrukcie do danej IR [14]. Ďalšou výhodou môže byť zjednodušenie aplikácie optimalizačných algoritmov. [22]

Do IR je možné preložiť jazyky vyššej, ako aj nižšej úrovne. Preklad z vyššej úrovne do nižšej sa zvykne označovať ako lowering [20] a naopak, preklad z nižšej do vyššej ako lifting. [21]

Ukážka liftovania assembly bloku *bb0* do bloku s pseudo IR kódom *irb0*, kde každá assembly inštrukcia je preložená na množinu operácii v IR, pričom jednotlivé operácie v množine na seba nevplývajú:

```
bb0:
    mov  r8, 5
    sub  ax, dx
    xchg ecx, edx
irb0:
    R8 = 5

    EAX[:16] = EAX[:16] - EDX[:16]
    zf = EAX[:16] - EDX[:16] == 0
    cf = EAX[:16] < EDX[:16]
    ...

    ECX = EDX
    EDX = ECX
```

Reaching definitions. Reaching definitions je technika analýzy toku dát programov určujúca, aké definície premenných dokážu dosiahnuť daný bod alebo riadok kódu.

Hovoríme, že premenná *v* v bode programu *d* dosiahne bod programu *u*, ak existuje cesta z *d* do *u*, ktorá neobsahuje definíciu *v*. [9]

Ukážka použitím bloku s pseudo kódom:

```
lbl0:
     0 A = 1
       B = 3
     1 B = 2
     2 A = A + B + 4
```

Ukážka reaching definitions bloku *lbl0*, z ktorej vieme vyčítať napríklad to, že inštrukciu číslo 2 bloku *lbl0* dosiahne premenná *A* definovaná inštrukciou číslo 0 v bloku *lbl0* a premenná *B* definovaná inštrukciou číslo 1 v bloku *lbl0*:

```
{
    (lbl0, 0) => {}
    (lbl0, 1) => {A: {(lbl0, 0)}, B: {(lbl0, 0)}}
    (lbl0, 2) => {A: {(lbl0, 0)}, B: {(lbl0, 1)}}
    (lbl0, 3) => {A: {(lbl0, 2)}, B: {(lbl0, 1)}}
}
```

[18]

Definition-Use chain. Definition-Use chain je dátová štruktúra, ktorá mapuje určitú definíciu danej premennej ku všetkým jej použitiam. [9]

Ukážka použitím bloku s pseudo kódom:

```
lbl0:
      0 A = 1
        B = 3
      1 B = 2
      2 A = A + B + 4
```

Definition-Use chain mapovanie:

```
{
     (lbl0, 0, A) => {(lbl0, 2, A)}
     (lbl0, 0, B) => {}
     (lbl0, 1, B) => {(lbl0, 2, A)}
     (lbl0, 2, A) => {}
}
```

[19]

Graf dátových závislostí. Graf dátových závislostí, alebo graf toku dát, reprezentuje závislosti medzi jednotlivými operáciami v kóde. Je to acyklický orientovaný graf, ktorého vrcholy predstavujú určité inštrukcie kódu a hrany dáta, ktoré do inštrukcií vstupujú alebo, naopak, vystupujú z nich. [7]

Hovoríme, že inštrukcia v bode programu *d* je závislá od inštrukcie v bode programu *u*, a teda existuje hrana (*d*, *u*) práve vtedy, keď ľubovoľná premenná *v* je definovaná inštrukciou v bode *u*, použitá inštrukciou v bode *d* a existuje cesta z *u* do *d*, ktorá neobsahuje definíciu *v*. [23]

Ukážka použitím bloku s pseudo kódom:

```
lbl0:
     0 A = 1
       B = 3
     1 B = 2 + B
     2 A = A + B + 4
```

Graf dátových závislostí s množinou vrcholov *V* a mapovaním hrán *H*, z ktorého vieme vyčítať napríklad, že premenná *A* definovaná inštrukciou číslo 2 v bloku *lbl0* je

priamo závislá od premennej *A* definovanej inštrukciou 0 v bloku *lbl0* a premennej *B* definovanej inštrukciou číslo 1 v bloku *lbl0*:

```
V =
{
    (lbl0, 0, A), (lbl0, 0, B),
    (lbl0, 1, B), (lbl0, 2, A)
};
H =
{
    {(lbl0, 0, A): ()},
    {(lbl0, 0, B): ()},
    {(lbl0, 1, B): ((lbl0, 0, B), )},
    {(lbl0, 2, A): ((lbl0, 0, A), (lbl0, 1, B))}
}
```

Symbolické vykonávanie. Symbolické vykonávanie je technika analýzy kódu programov, v ktorej sa reprezentujú určité premenné symbolickými hodnotami namiesto konkrétnymi dátami a pri ľubovoľných operáciách s takýmito premennými vznikajú symbolické výrazy. [1]

Stav symbolicky vykonávaného programu obsahuje hodnoty všetkých premenných programu, čítač inštrukcií a zhromaždené obmedzenia, ktoré na začiatku zvolené symbolické hodnoty musia splniť, aby sa program počas štandardného behu dostal z nejakého počiatočného miesta na aktuálnu pozíciu. [1]

Zhromaždené obmedzenia a počiatočné hodnoty je možné chápať ako teóriu v logike a na to, aby sme získali konkrétne hodnoty vstupných premenných, potrebujeme nájsť ľubovoľný model, čo je možné dosiahnuť pomocou vybraného SMT solver-u. [2]

Dynamické symbolické vykonávanie. Dynamické symbolické vykonávanie, alebo DSE (dynamic symbolic execution a zvykne sa tiež nazývať concolic execution), je symbolické vykonávanie sprevádzané konkrétnym. DSE sa vždy sústredí iba na jeden konkrétny beh. Cestou zbiera obmedzenia pre počiatočné symbolické hodnoty a potom, ako skončí, sa pokúsi nájsť inverziu takého obmedzenia, pomocou ktorého dokáže nájsť doposiaľ nenavštívenú časť kódu. [5]

DSE nás vie ušetriť viacerých nerestí čistého symbolického vykonávania, ako napríklad veľkého rozmachu možných ciest vznikajúcich v slučkách, príliš zložitých výrazov nahradením konkrétnymi hodnotami alebo získaním konkrétnych hodnôt po

systémovom alebo knižničnom volaní. Na druhú stranu môže DSE však stratiť niektoré možné, no komplikované, cesty. [5]

1.1.3 Iné teoretické východiská

Posledná podkapitola z teoretických východísk pokryje zvyšné pojmy z viacerých oblastí, ktoré sa štandardne nezvyknú používať a nebolo ich dosť na to, aby malo význam pre ne vytvárať ďalšie samostatné podkapitoly.

Vzdialené volanie procedúry. Vzdialené volanie procedúry, alebo RPC, poskytuje netradičný spôsob prístupu k sieťovým službám. Namiesto klasického prístupu pomocou odosielania a prijímania správ klient volá lokálnu procedúru, ktorá skrýva detaily sieťovej komunikácie.

Pri takomto volaní sa prostredie klienta pozastaví, parametre danej procedúry sa presunú po sieti do prostredia, kde sa má daná procedúra spustiť. Procedúra sa spustí a jej výsledky sa presunú späť do pôvodného prostredia, v ktorom pokračuje vykonávanie. [12]

Skok. Pod skokom chápeme inštrukciu vetvenia, teda takú inštrukciu, po ktorej môže byť vykonaná aj iná ako nasledujúca inštrukcia v pamäti.

Základný blok. Pod základný blokom chápeme maximálnu sériu inštrukcií, ktorú neprerušujú žiadne skoky a sama ich neobsahuje, s výnimkou poslednej inštrukcie.

Cesta. Pod cestou rozumieme z hľadiska toku vykonávania zoznam pozostávajúci zo základných blokov v takom poradí, v akom za sebou nasledovali počas určitého behu programu.

Graf toku riadenia. Pod grafom toku riadenia rozumieme orientovaný graf, ktorého vrcholy reprezentujú základné bloky a hrany reprezentujú skoky v toku riadenia. [3]

1.2 Technologické východiská

V tejto kapitole si priblížime technologické východiská bakalárskej práce a popíšeme užitočné funkcie, ktoré budú použité pri implementácii.

1.2.1 Miasm

Miasm je open source framework pre reverzné inžinierstvo. Má množstvo rôznych využití nakoľko obsahuje pestrú paletu funkcií, ktorá mu umožňuje:

- Assemblovať a disassemblovať inštrukcie mnohých architektúr: X86, ARM, MIPS, SH4 a MSP430.
- Otvárať, modifikovať a generovať spustiteľné súbory pre operačné systémy Microsoft Windows – PE (Portable Executable) a Linux – ELF (Executable and Linkable Format).
- Reprezentovať sémantiku jazyku assembler pomocou Miasm IR, ktorú je možné exportovať do LLVM IR.
- Emulovať kód prostredníctvom JIT (just-in-time compiler).
- Zjednodušovať symbolické výrazy pre automatickú deobfuskáciu určitých obfuskačných techník.
- Aplikovať na kód rôzne grafové algoritmy a techniky analýzy kódu programov.
- Symbolicky vykonávať vybrané časti kódu.
- Použiť nad vybraným kódom dynamické symbolické vykonávanie.
- Pomocou rozšíreného algoritmu dynamického symbolického vykonávania automaticky invertovať obmedzenia podľa zvolenej stratégie tak, aby dosiahol vždy nový a nepokrytý základný blok, skok alebo cestu. [5]

Napriek tomu má však Miasm stále niekoľko nedostatkov, ako napríklad to, že nedokáže:

- spracovať jump tabuľky;
- uvažovať nad celým programom ako celkom;
- rozlíšiť funkcie bez návratu;
- sám zistiť výšku zásobníku po volaní ľubovolnej funkcie.

Z týchto dôvodov býva potrebné opierať sa o nejaký v určitých oblastiach pokročilejší nástroj ako pomocnú barličku. [17]

Ďalej popíšeme, ako vykonať niektoré základné operácie v Miasm-e a čo sa za nimi skrýva, v nami použitej verzii.

Disassemblovanie. Pomocou nasledujúceho kódu dokážeme disassemblovať súbor *file.bin* na adrese jeho entry point-u, ktorú volíme na riadku 16 ako parameter metódy *dis_multiblock*. S disassemblovaným kódom môžeme ďalej pracovať prostredníctvom inštancie *asmcfg* triedy miasm.core.asmblock.AsmCFG.

Konštruktor triedy *Container* na riadku 8 automaticky deteguje formát daného súboru, spracuje ho a sprístupní rôzne informatívne údaje, ako napríklad použitú architektúru alebo adresu entry point-u, ktoré využijeme na riadkoch 9 a 16. Na riadku 9 inicializujeme inštanciu triedy Machine, ktorá predstavuje abstraktnú továreň zjednodušujúcu prácu s triedami závislými od architektúry.

Najzaujímavejší je riadok 11, kde inicializujeme objekt *mdis* bázovej triedy *miasm.core.asmblock.disasmEngine* pomocou metódy *dis_engine*, ktorej jediný požadovaný parameter je binárny prúd dát, s ktorým bude pracovať. Okrem toho je možné použiť voliteľné parametre:

- *loc_db* využívajúci konkrétnu inštanciu triedy *LocationDB*, ktorej úloha je uchovávať informácie týkajúce sa určitej adresy v kóde namiesto vytvárania novej.
- *dont_dis* obsahujúci adresy, po ktorých ďalší kód už nemá byť disassemblovaný.

Na nasledujúcom, jedenástom, riadku nastavujeme spätné volanie *dis_block_callback*, ktoré je volané po každom disassemblovanom bloku, pričom jeho parametre sú samo vysvetľujúce.

```
1. ...
2. from miasm.analysis.binary import Container
3. from miasm.analysis.machine import Machine
4. from miasm.core.locationdb import LocationDB
5.
6. def dis_callback(mdis, cur_block, offsets_to_dis):
7.         return
8.
9. cont = Container.from_stream(open('file.bin', 'rb'))
10. machine = Machine(cont.arch)
11. unreachable = []
12. mdis = machine.dis_engine(cont.bin_stream,
13.                           loc_db=LocationDB(),
14.                           dont_dis=unreachable)
15. mdis.dis_block_callback = dis_callback
16. asmcfg = mdis.dis_multiblock(cont.entry_point)
```

Obr. 1: Ukážka disassemblovania kódu pomocou Miasm-u.

Assemblovanie. V nasledujúcom kóde zassembujeme upravený kód na adresu *out_addr*, pričom neprekročíme adresu *max_addr*. No assemblovanie obnáša niekoľko problémov, ktoré nemusia byť hneď jasné a nevyskytujú sa pri disassemblovaní.

Prvým z problémov je to, že pri zmene adresy výsledných inštrukcií je pri 64 bitovej architektúre potrebné dávať pozor na to, aby sa inštrukcie pristupujúce k dátam RIP-relatívnym adresovaním správne prepočítali.

Ďalej je nutné dávať pozor na to, aby adresy aktuálne prislúchajúce k blokom inštrukcií boli odstránené, pretože pri opätovnom zassemblovaní môžu skončiť na inej adrese, čo by viedlo ku konfliktu, nakoľko žiadny blok nemôže byť na viacerých adresách súčasne.

Na záver získame na riadku 20 asociatívne pole *patches* priraďujúce adresy k novým dátam. Aplikovanie týchto zmien na daný súbor alebo jeho kópiu je už jednoduchý posledný krok.

Tiež je potrebné spomenúť, že:

- *ExprId* je trieda predstavujúca ľubovoľný identifikátor v Miasm IR.
- Pod *ExprId('RIP', 64)* chápeme register RIP o veľkosti 64 bitov.
- *new_next_addr_card* je špeciálny identifikátor, ktorý bude počas fázy assemblovania nahradený adresou nasledujúcej inštrukcie.
- Objekt *mn* z *machine* poskytuje metódy umožňujúce assemblovať a disassemblovať jednotlivé inštrukcie danej architektúry. Jeho bázová trieda je *miasm.core.asmblock.cls_mn*.

```
1. ...
2. from miasm.expression.expression import ExprId, ExprLoc
3. from miasm.core.interval import interval
4. from miasm.core.asmblock import asm_resolve_final
5.
6. rip = ExprId("RIP", 64)
7. name_loc = asmcfg.loc_db.get_or_create_name_location('_')
8. new_next_addr_card = ExprLoc(name_loc, 64)
9. for block in asmcfg.blocks:
10.        for instr in block.lines:
11.            for ind in range(len(instr.args)):
12.                if rip in instr.args[ind]:
13.                    next_addr = ExprInt(instr.offset + instr.l, 64)
14.                    fix_dict = {rip: rip + next_addr - new_next_addr_card}
15.                    instr.args[ind] = instr.args[ind].replace_expr(fix_dict)
16.        if asmcfg.loc_db.get_location_offset(block.loc_key):
17.                asmcfg.loc_db.unset_location_offset(block.loc_key)
18.
19. asmcfg.loc_db.set_location_offset(asmcfg.heads()[0], out_addr)
20. patches = asm_resolve_final(machine.mn, asmcfg,
21.                             asmcfg.loc_db,
22.                             dst_interval=interval([(out_addr, max_addr)]))
```
Obr. 2: Ukážka assemblovania kódu pomocou Miasm-u.

Symbolické vykonávanie. V ďalšej ukážke pokračujeme s kódom z predchádzajúcich ukážok a symbolicky vykonáme už disassemblovaný úsek kódu, počnúc entry pointom.

Objekt *ira* z *machine* použitý na riadku 5 poskytuje metódy umožňujúce liftovať jednotlivé disassemblované inštrukcie danej architektúry do IR Miasm-u.

ira je rozšírením triedy *ir*, ktoré inštrukcie volaní procedúr neprekladá doslovne, teda napríklad pri architektúre x86 ich nevníma ako kombináciu inštrukcií *push* a *jump*, ale predpokladá, že vykonávanie bude neskôr pokračovať na nasledujúcej inštrukcii.

Registre, ktoré majú byť clobbered podľa ABI, a smerník na zásobník sú pritom označené ako poškvrnené týmto volaním, čo umožňuje pracovať so samostatnou funkciou bez toho, aby sme rekurzívne spracovávali ďalšie.

Nasledujúcim zaujímavým riadkom je riadok 8, na ktorom symbolu predstavujúcemu register *EDX* pred začiatkom symbolického vykonávania priradíme 32 bitovú číselnú hodnotu 123.

Posledným krokom je spustenie samotného symbolického vykonávania na riadku 9, ktoré vykonáva symbolicky blok za blokom. Vykonávanie končí, keď je adresa nasledujúceho bloku podľa aktuálneho symbolického stavu nejednoznačná.

```
1. ...
2. from miasm.ir.symbexec import SymbolicExecutionEngine
3. from miasm.expression.expression import ExprInt
4.
5. ira = machine.ira(LocationDB())
6. ircfg = ira.new_ircfg_from_asmcfg(asmcfg)
7. sb = SymbolicExecutionEngine(ira)
8. sb.symbols[machine.mn.regs.EDX] = ExprInt(123, 32)
9. symbolic_pc = sb.run_at(ircfg, cont.entry_point)
```

Obr. 3: Ukážka symbolického vykonávania pomocou Miasm-u.

Emulácia pomocou Sandbox-u. V prípade, že chceme emulovať časti klasického spustiteľného súboru a nie shell-kód, môžeme využiť jednu z tried *Sandbox* vybranú podľa potrebného OS a architektúry. Táto trieda za nás správne načíta súbor do pamäte a umožní s ním okamžite pracovať.

V našom prípade zvolíme OS Windows a architektúru *x86_32* pre emulovanie súboru *file.bin*, o ktorom už vieme, že je danej architektúry a určený pre OS Windows. Vyhovujúci *Sandbox* je teda *Sandbox_Windows_x86_32*, v ktorom nastavujeme viaceré nastavenia ako:

- *mimic_env*, ktoré povolí načítanie počiatočného prostredia spustiteľného súboru pred samotným behom. Načíta do pamäte napríklad premenné prostredia a argumenty.
- *use_windows_structs*, ktoré povolí simuláciu štandardných Windows štruktúr ako PEB, LDR alebo SEH.
- *usesegm*, ktoré povolí simuláciu segmentov.
- *jitter*, ktorý vyberie konkrétny jitter, LLVM jitter je zvolený kvôli tomu, že podporuje 128 bitové operácie a je rýchlejší ako Python jitter.

Tretí a posledný parameter daného sandbox-u je nepovinný a obsahuje slovník priraďujúci názvy API funkcií v tvare 'xxx_%meno%' k implementovaným funkciám, ktoré majú byť spustené namiesto nich, v našom prípade *globals()*.

```
1. ...
2. from miasm.analysis.sandbox import Sandbox_Win_x86_32
3.
4. parser = Sandbox_Win_x86_32.parser()
5. options.mimic_env = True
6. options.use_windows_structs = True
7. options.usesegm = True
8. options.jitter = 'llvm'
9.
10. sb = Sandbox_Win_x86_32('file.bin', options, globals())
11. sb.run()
```

Obr. 4: Ukážka emulácie kódu pomocou Miasm-u

Dynamické symbolické vykonávanie. Samotné dynamické symbolické vykonávanie dokážeme k sandbox-u pripojiť už veľmi jednoducho.

V ukážke pred spustením uložíme na riadku 6 aktuálny stav.

Na riadku 7 okrem toho, podobne ako v sandbox-e, registrujeme funkcie, ktoré sa majú vykonať pri volaní určitého API volania. Názvy funkcií sú vyjadrené analogicky s tým, že na konci obsahujú ešte '_symb', a teda 'xxx_%meno%_symb'.

Pri triede *DSEEngine* je ďalej potrebné spomenúť, že umožňuje jednoducho symbolizovať určité rozsahy pamäti, upravovať symboly a aktualizovať symbolický stav podľa konkrétneho.

Po spustení sandbox-u na riadku 8 iba obnovíme pôvodný stav, čím zrušíme všetky zmeny, ktoré nastali počas volania.

```
1. ...
2. from miasm.analysis.dse import DSEEngine
3.
4. dse = DSEEngine(sb.machine)
5. dse.attach(sb.jitter)
6. initial_snap = dse.take_snapshot()
7. dse.add_lib_handler(sb.libs, globals())
8. sb.run()
9. dse.restore_snapshot(initial_snap)
```

Obr. 5: Ukážka pripájania dynamického symbolického vykonávania k Sandbox-u v Miasm-e

Graf dátových závislostí. Graf dátových závislostí v Miasm-e predstavuje trieda *DependencyGraph*. Pri inicializácii tejto triedy jej je možné posunúť niekoľko voliteľných parametrov:

- *implicit* – značka, ktorá keď je nastavená, symbolický čítač inštrukcií je pridaný medzi sledované výrazy pri každom prechádzanom bloku kódu.
- *apply_simp* – značka, ktorá keď je nastavená, na spracovávané výrazy sú aplikované zjednodušujúce pravidlá. Teda napríklad v prípade, že sa sleduje register *EAX* a narazí sa na inštrukciu *XOR EAX, EAX*, bude sa vedieť, že do registra *EAX* bude vždy priradená nula a nie je potrebné ďalej sledovať *EAX*.
- *follow_mem* – značka, ktorá keď je nastavená, operácie s pamäťou a všetky výrazy, od ktorých tieto operácie závisia, sú pridané medzi sledované.
- *follow_call* – značka, ktorá keď je nastavená, všetky volania a výrazy, od ktorých závisia tieto volania, sú pridané medzi sledované.

Metóda *get()* následne generuje výsledky pre dané elementy konkrétneho riadku, pričom posledný parameter je zoznam predstavujúci bloky, na ktorých by mal algoritmus zastať.

Každý výsledok metódy *get()* predstavuje jednu acyklickú cestu toku dát, ktorá ovplyvňuje dané elementy na konkrétnom riadku. Graf výsledku je možné vygenerovať pomocou metódy *as_graph()*.

Aplikovaním metódy *emul()* vieme získať hodnoty pre hľadané elementy podľa aktuálneho grafu, ktoré však môžu byť aj symbolické. V prípade, že bol nastavený parameter *implicit*, dokážeme metódou *is_satisfiable()* zistiť, či existuje nejaký model pre dané obmedzenia. Ak nejaký model existuje, vieme ľubovoľný model metódou *consraints()* aj vygenerovať.

Definition-Use chain. Štruktúru definition-use chain je možné získať jednoducho pomocou triedy *miasm.analysis.data_flow.DiGraphDefUse*, ktorej potrebujeme poskytnúť akurát reaching definitions vygenerované triedou *miasm.analysis.data_flow.ReachingDefinitions*.

Dominátor. Všetci dominátory môžu byť nad určitou inštanciou triedy *miasm.core.graph.DiGraph* naraz vypočítaní jednoducho metódou *compute_dominators()*.

Okamžitý dominátor. Všetci okamžití dominátory môžu byť nad určitou inštanciou triedy *miasm.core.graph.DiGraph* naraz vypočítaní jednoducho metódou *compute_immediate_dominators()*.
[6]

1.2.2 IDAPython

IDAPython je plugin pre známy multi-platformový disassembler IDA, ktorý umožňuje písať skripty pre tento disassembler pomocou programovacieho jazyku Python a poskytuje úplný prístup k IDA API.

IDA, narozdiel od Miasm-u, spracuje vo väčšine prípadov celý program a môže mu dopomôcť napríklad poukázaním na funkcie bez návratu alebo poskytnutím hĺbky zásobníku v ľubovoľnom bode programu. Teda aspoň za predpokladu, že sama bola schopná spracovať program bez problémov. Na druhú stranu chýba IDE väčšina funkcionality, ktorá sa v Miasm-e nachádza.

Nakoľko IDAPython obsahuje obrovské množstvo funkcií a budeme potrebovať iba niekoľko z nich, popíšeme iba tie konkrétne:

- *idautils.Functions(start=None, end=None)* – vráti zoznam všetkých funkcií medzi danými adresami.
- *idaapi.get_input_file_path()* – vráti cestu spracovávaného súboru.
- *idaapi.get_func(ea)* – vráti objekt reprezentujúci funkciu na adrese *ea*.
- *idc.get_operand_value(ea, n)* – vráti hodnotu n-tého operandu inštrukcie na adrese *ea*.
- *idaapi.get_arg_addrs(caller)* – vráti zoznam adries volania na adrese *caller*.
- *idc.get_spd(ea)* – vráti rozdiel medzi hodnotou smerníku na zásobník na začiatku funkcie a na adrese *ea*.
- *idaapi.reload_file(file, is_remote)* – opätovne načíta súbor *file* z disku, *is_remote* určuje, či je daný súbor uložený lokálne alebo na vzdialenom počítači so spusteným vzdialeným debugovacím serverom.
- *ida_funcs.add_func(ea1, ea2=BADADDR)* – pridá novú funkciu s rozsahom od adresy *ea1* po *ea2*. Ak je *ea2* rovné ‘BADADDR‘, pokúsi sa rozlíšiť koniec funkcie samostatne.
- *idaapi.FUNC_LIB* – značka určujúca, či je daná funkcia knižničná.

[16]

1.2.3 RPyC

RPyC, alebo Remote Python Call, je knižnica pre programovací jazyk Python. Slúži predovšetkým na symetrické vzdialené volania procedúr, ktoré umožňujú pracovať s vzdialenými objektmi tak, ako keby boli lokálne.

Pomocou RPyC sa dokážeme napríklad jednoducho pripojiť z ľubovoľného Python interpreteru k inej Python inštancii, napríklad takej, ktorá obsahuje moduly IDAPython.

Keďže RPyC je rozsiahly projekt a potrebujeme iba zlomok jeho funkcionality, popíšeme iba tú. Nasledujúca ukážka spustí server, ktorý čaká na jediné pripojenie a po jeho nadviazaní a ukončení už ďalej neočakáva ďalšie. *SlaveService* umožňuje importovať a spúšťať po pripojení na server ľubovoľný kód.

```
1. from rpyc.utils.server import OneShotServer
2. from rpyc.core import SlaveService
3.
4. server = OneShotServer(SlaveService, hostname="127.0.0.1",
5.                        port=168, reuse_addr=True, ipv6=False,
6.                        authenticator=None,
7.                        auto_register=False)
8. server.start()
```

Obr. 6: Ukážka inicializácie RPyC servera

V ďalšej ukážke uvidíme, ako sa na spustený server pripojiť a vykonávať na ňom základné operácie. Konkrétne vypíšeme ‘hi there‘ na strane servera riadkom 5. Vymeníme štandardný výstup na strane servera za štandardný výstup klienta. Následne zopakujeme príkaz s výrazom ‘hi here‘ na riadku 7 a tento krát sa výraz zobrazí u klienta. Nakoniec zavrieme spojenie.

```
1. import rpyc
2. import sys
3.
4. conn = rpyc.classic.connect("127.0.0.1", 168)
5. conn.execute("print('hi there')")
6. conn.modules.sys.stdout = sys.stdout
7. conn.execute("print('hi here')")
8. conn.close()
```

Obr. 7: Ukážka pripojenia k RPyC serveru

[6, 15]

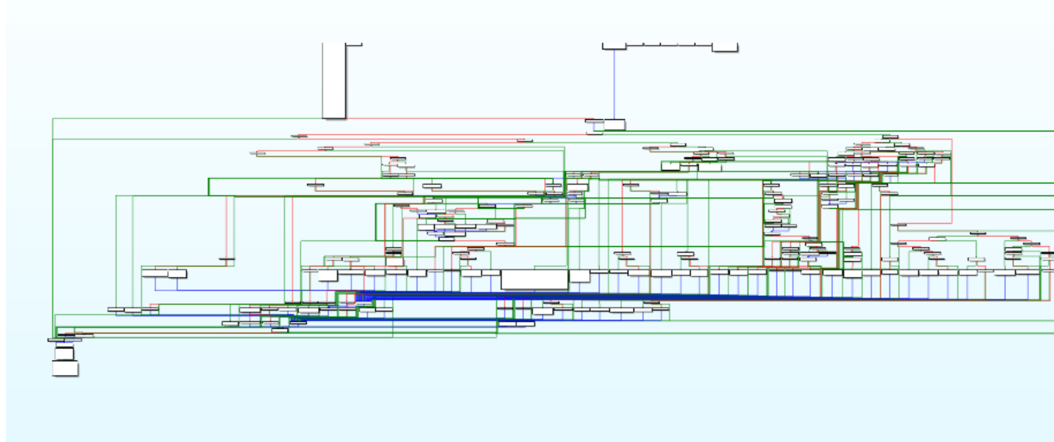
2 Analýza obfuskačných techník

V nasledujúcich kapitolách sa budeme venovať analýze obfuskačných techník, ktoré boli aplikované na komponenty skupiny Stantinko. Popíšeme aký dopad na kód majú tieto techniky a z toho sa pokúsime vydedukovať, ako by mohli fungovať.

V ďalších kapitolách následne použijeme tieto zistenia na to, aby sme usúdili, či sa jedná o už známe techniky a navrhli spôsob, ako ich prekonať.

2.1 Prvý kontakt

Analýzu začneme letmým pohľadom na niekoľko funkcií nachádzajúcich sa v blízkosti začiatku jedného z obfuskovaných programov. Funkcie na prvý pohľad vyzerajú byť neštandardne komplikované. Čo sa týka počtu základných blokov a tvaru, tak priemerný graf toku riadenia funkcií vyzerá približne nasledovne:



Obr. 8: Ukážka bežného grafu toku riadenia obfuskovaných funkcií.

Pokiaľ by sa nám nepodarilo zjednodušiť tieto funkcie, analýza kódu by bola zdĺhavá, náchylná na chyby a vždy by ju bolo potrebné pracne opakovať.

2.2 Obfuskácia toku riadenia

Po bližšom preskúmaní viacerých vyššie spomenutých komplikovaných funkcií sme zistili, že obsahujú opakujúce sa konštrukty, ktoré obfuskujú tok riadenia programu. Bývajú aplikované na ľubovolné miesta a vnárajú sa aj do seba.

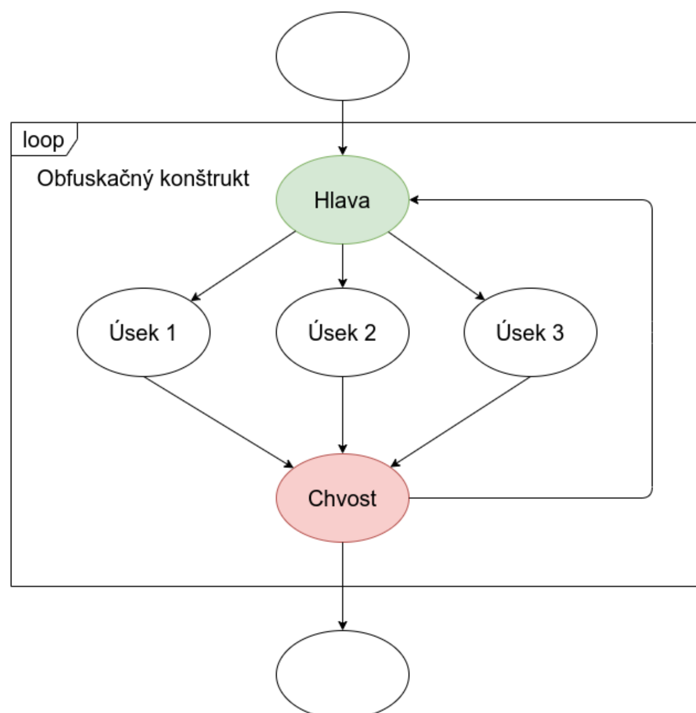
Tieto konštrukty sa skladajú z cyklu, ktorý začína základným blokom *hlava* s príkazom switch. Switch sa riadi premennou, ktorú budeme označovať *riadiaca premenná*.

Riadiaca premenná je buď jedným z parametrov danej funkcie, alebo definovaná a inicializovaná pred takýmto cyklom.

Spomínaný príkaz switch vykoná na základe *riadiacej premennej* jeden z úsekov kódu, ktoré budeme označovať jednoducho *úsek*.

Úseky predstavujú množiny základných blokov končiacich jediným blokom, ktorý budeme označovať ako *chvost*. Zvyšuje sa tu hodnota *riadiacej premennej* o určitú konštantu a rozhoduje, či pokračovať v cykle.

Takýto konštrukt môžeme zobraziť následovne:



Obr. 9: Ukážka štandardného obfuskačného konštruktu.

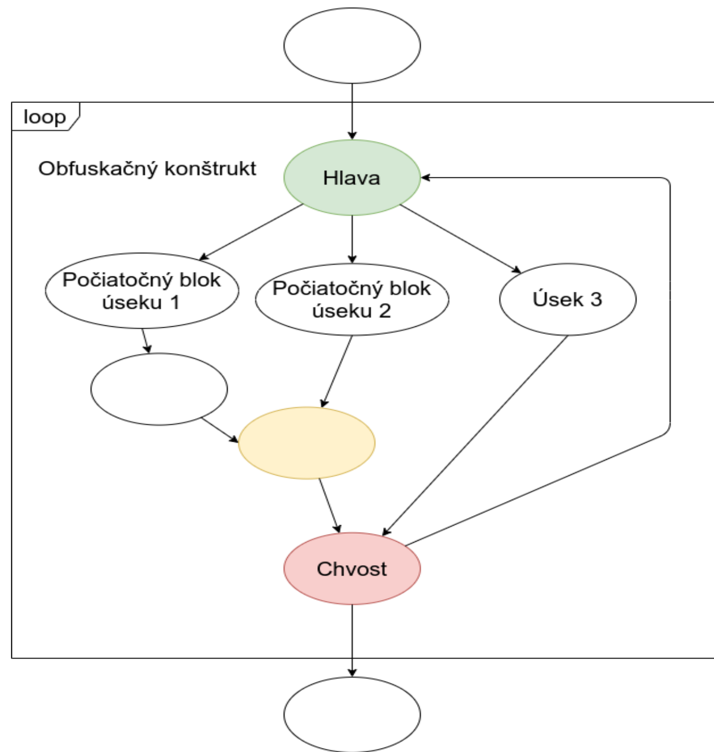
Tieto príkazy switch sú v našom prípade na úrovni asssembler-u reprezentované ako kombinácia jump tabuliek a binárneho vyhľadávania. Čo značí, že po aplikovaní tejto obfuskácie ešte prebehli štandardné optimalizácie kompilátoru. Tým pádom je táto obfuskácia s vysokou pravdepodobnosťou aplikovaná na vyššej úrovni, ako je assembler.

No daný konštrukt nemá vždy presne takúto podobu a zvykne obsahovať anomálie, ktoré sú s vysokou pravdepodobnosťou výsledkom optimalizácií použitého kompilátoru, keďže manuálne vytvárať niečo také by nemalo zmysel.

Nakoľko je počet a dopad týchto anomálií veľký, s vysokou istotou predpokladáme, že obfuskačná technika bola aplikovaná na samotný zdrojový kód a nie ako transformačný priechod IR použitého kompilátoru.

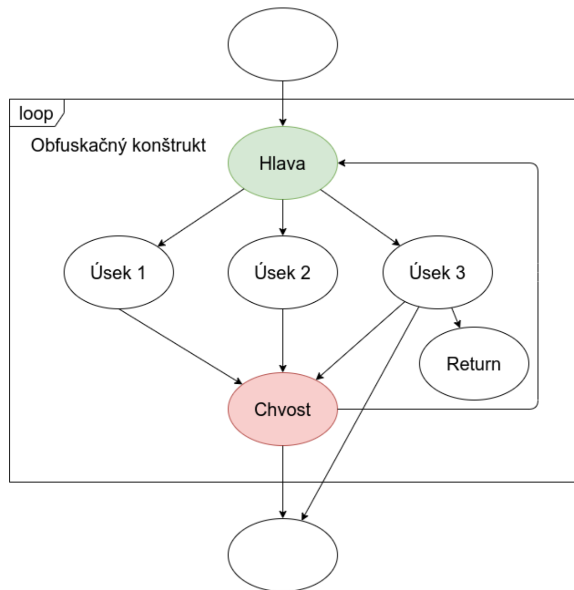
Počas analýzy sme videli nasledujúce anomálie, ktoré sa medzi sebou môžu kombinovať:

1. Niektoré *úseky* sú mŕtvy kód – nikdy nebudú spustené. Napríklad hodnota *riadiacej premennej* prislúchajúcej týmto úsekom môže byť ľubovolné párne číslo, pričom jej počiatočná hodnota je nepárna a konštanta, o ktorú sa zvyšuje, je párna.
2. *Úseky* sa môžu prekrývať – môže existovať neprázdna množina základných blokov patriacich súčasne dvom ľubovolným *úsekom*.



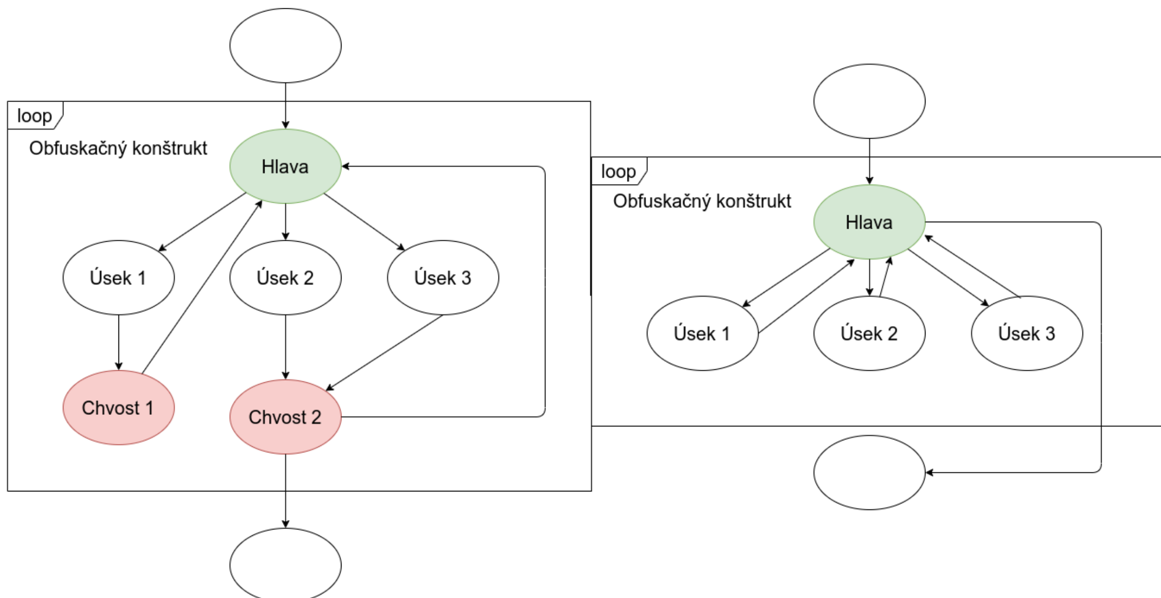
Obr. 10: Ukážka obfuskačného konštruktu s prekryvom základných blokov (zvýraznené žltou).

3. *Úseky* sa môžu vrátiť z funkcie predčasne alebo preskočiť chvost, teda prejsť priamo na ďalší blok mimo obfuskačného konštruktu.



Obr. 11: Ukážka predčasného návratu z obfuskačného konštruktu (v úseku 3).

4. *Chvosty* môžu byť viaceré alebo, naopak, žiadne – v druhom prípade je *riadiaca premenná* zvyšovaná na konci každého úseku.



Obr. 12: Ukážka obfuskačného konštruktu s viacerými chvostami (vľavo) a bez chvostov (vpravo).

5. Hodnota *riadiacej premennej* môže byť použitá v úsekoch s legitímnym kódom – v deobfuskovanom kóde musí byť jej hodnota uchovaná, aby sme nezmenili význam pôvodného kódu.
6. Niekedy sú v *hlave* a *chvostoch* aj inštrukcie, ktoré nie sú súčasťou obfuskačného konštruktu a potrebujeme ich zachovať, aby sme nezmenili význam pôvodného kódu.
7. Existujú prípady, v ktorých neexistuje úsek, ktorému prislúcha aktuálna hodnota *riadiacej premennej*. V takom prípade sa prejde priamo na *chvost*.
8. Stáva sa, že sú použité dve zosynchronizované *riadiace premenné*. Jedna pre *hlavu*, ktorá určuje, aký úsek bude vykonaný. Druhá pre *chvost,* ktorá rozhodne, či prerušiť slučku.

[8]

2.3 Mŕtvy kód

Pod mŕtvym kódom myslíme kód, ktorý nie je dosiahnuteľný, teda nemôže byť nikdy spustený. Mŕtvy kód bol spomenutý už vyššie pri obfuskácii toku riadenia, ale toho sa dokážeme zbaviť deobfuskáciou toku riadenia.

Okrem toho sa používajú aj exporty, reťazce a resource súbory, ktoré nie sú nikdy použité, teda tiež predstavujú formu mŕtveho kódu bez vplyvu na funkcionalitu. Takéto dáta a exporty nie je jednoduché rozlíšiť a nakoľko samé o sebe nekomplikujú poznateľne samotnú analýzu, nie je potrebné zaoberať sa ich deobfuskáciou. [8]

2.4 Irelevantný kód

Priamo medzi riadky samotného kódu býva primiešaný aj kód, ktorý nemá žiadny vplyv na funkcionalitu. Na rozdiel od mŕtveho kódu je tento dosiahnuteľný a môže byť normálne spustený. Nevyskytuje sa však až v takej miere, že by robil analýzu poznateľne náročnejšiu, jeho deobfuskácii sa teda neskôr venovať nebudeme.

```
hWnd = WindowFromPoint(0i64);
sub_10043090(&v60, strlen((const char *)&v60));
hDC = GetDC(hWnd);
v14 = CreateServiceA(0, ServiceName, DisplayName, 0xF01FFu, 0x10u, 3u, 1u, 0, 0, 0, 0, 0, 0);
sub_10043090(v46, strlen(v46));
if ( hDC )
{
  ReleaseDC(hWnd, hDC);
  hDC = 0;
  v15 = Src;
}
v16 = 0;
if ( v14 )
  v16 = v37;
v37 = v16;
if ( v16 )
{
  v37 = v16;
  StartServiceA(v14, 0, 0);
  DeleteService(v14);
  CloseServiceHandle(v14);
  v15 = Src;
}
CancelDC(hDC);
```

Obr. 13: Ukážka irelevantného kódu.
[8]

2.5 Obfuskácia reťazcov

Všetky na prvý pohľad viditeľné reťazce nie sú samé o sebe priamo použité k ničomu. Nie je jasné odkiaľ pochádzajú, keďže sú v rôznych jazykoch a líšia sa aj obsahom. V kóde sa používajú na poskladanie reťazcov, ktoré budú skutočne použité, prípadne sa s nimi nerobí nič.

Skutočné reťazce sa zložia za behu zo spomenutých nastražených reťazcov a samostatných písmen. Na skladanie bývajú použité štandardné C funkcie pre prácu s reťazcami, ako napríklad `strcpy()`, `strcat()`, `strncat()`, `sprintf()`, `memmove()` a ich unikódové verzie. Taktiež sa zvyknú priamo presúvať skupiny byte-ov.

```
strcpy(String1, "{");
lstrcpyA(&String1[1], "\"met");
strcpy(v13, "hod\":\"su");
lstrcpyA(&v13[8], "b");
strcpy(&v13[9], "m");
strncat(String1, "it\"", 3u);
memmove(&Dst, ",", 2u);
lstrcpyA(v15, "\"p");
strncat(String1, "a", 1u);
strcpy(v16, "rams\"");
memmove(&v16[5], &unk_1007AFE4, 3u);
memmove(&v16[7], &unk_1007AFE8, 4u);
lstrcatA(String1, "\"");
memmove(v17, ":\"", 3u);
v6[0] = 34;
*(_WORD *)&v6[1] = 8748;
```

Obr. 14: Ukážka skladania reťazca.

[8]

3 Popis podobných obfuskačných techník

V tretej časti práce prirovnáme techniky popísané v predchádzajúcej kapitole k podobným a už zdokumentovaným technikám.

3.1 Vyrovnávanie toku riadenia

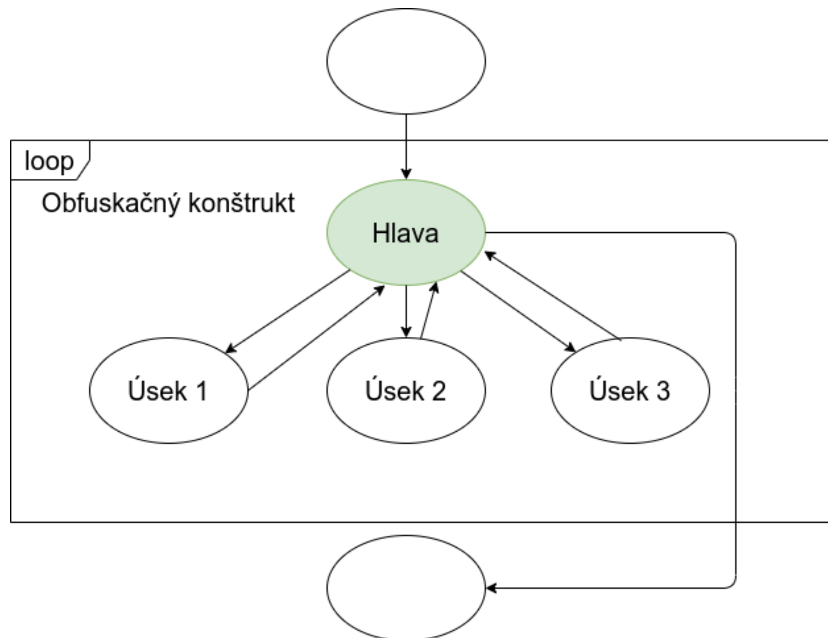
Vyrovnávanie toku riadenia je obfuskačná technika, ktorá do veľkej miery pripomína spôsob, akým bol obfuskovaný tok riadenia v našom prípade, považujeme ho teda za variáciu tejto techniky.

Vyrovnanie toku riadenia je možné bežne docieliť rozdelením cieľovej funkcie na *bloky*. Tieto *bloky* sa následne použijú ako case prípady príkazu switch nachádzajúceho sa v slučke. Príkaz switch na základe *riadiacej premennej* určí, aký *blok* bude vykonaný.

Každému *bloku* je počas rozdelenia priradené *ID. Riadiaca premenná* je na začiatku nastavená na *ID* prvého *bloku*.

Všetky bloky na záver nastavujú hodnotu *riadiacej premennej* na *ID* nasledujúceho *bloku*. V prípade, že určitý blok končí podmieneným skokom, hodnota *riadiacej premennej* `X` bude nastavená podmieneným priradeným v súlade s danou podmienkou, teda napríklad ako `mov X, ID1; cmp …, …; cmova X, ID2`.

Anomáliu s chýbajúcimi chvostami z popisu obfuskácie toku riadenia môžeme považovať priamo ako prípad vyrovnávania toku riadenia. Táto anomália presne sedí na jeho popis – v jej grafe treba iba brať *úseky* ako *bloky*:



Obr. 15: Ukážka vyrovnaného toku riadenia.

Vo všeobecnosti je na deobfuskáciu tejto základnej formy potrebné identifikovať takéto slučky, extrahovať riadiacu premennú a na základe nej usporiadať dané *bloky*.

Existujúce riešenia nedokážu deobfuskovať implementáciu vyrovnávania toku riadenia použitú v kóde skupiny Stantinko z viacerých možných dôvodov:

- Nepočítajú s vyššie popísanými anomáliami spôsobenými optimalizáciami kompilátoru – väčšina variácií vyrovnávania toku riadenia, na ktoré boli také riešenia cielené, je implementovaná ako transformačný priechod IR určitého kompilátoru v neskoršej fáze kompilácie.
- *Bloky* pozostávajú z dlhších úsekov kódu, ktoré môžu obsahovať aj inštrukcie vetvenia.

- Časti *blokov* sa môžu prekrývať.
- *Riadiaca premenná* je zvyšovaná o konštantu a bežne to nie je na konci každého *bloku*.
- Bežne sa takéto obfuskačné konštrukty vyskytujú v každej funkcii iba raz a už vôbec sa nevnárajú do seba.
- *Riadiaca premenná* nebýva inicializovaná pomocou parametra funkcie.
- Bežne sa nespájajú viaceré funkcie do jednej.

[8]

3.2 Mŕtvy kód

Pridávanie mŕtveho kódu je štandardná obfuskačná technika, ktorá podľa definície spočíva v pridaní nedosiahnuteľného kódu, teda takého, ktorý sa nedokáže za bežných okolností vykonať. Mŕtvy kód môže napríklad nasledovať za podmienkou, ktorá nemôže byť nikdy splnená. [3]

```
1. def priklad_mrtvy_kod(a: int):
2.         if a == a-1:
3.                 ...  # nenastane
4.         return (a & 0xFF) ^ 0xFF
```

Obr. 16: Ukážka mŕtveho kódu.

V našom prípade mŕtvy kód použitý s vyššie popísanou obfuskáciou toku riadenia, spadá pod túto definíciu a vieme ho efektívne eliminovať deobfuskáciou techniky, s ktorou bol použitý. Nevyužité reťazce, exporty a resource súbory sa nepoužívajú bežne.

3.3 Irelevantný kód

Pridávanie irelevantného kódu je štandardná obfuskačná technika, ktorá podľa definície spočíva v pridaní kódu bez vplyvu na správanie programu, ktorý sa narozdiel od mŕtveho kódu vykoná. Používajú sa najmä inštrukcie, ktoré majú opačné správanie

a navzájom svoje efekty nulujú, prípadne volania funkcií, ktorých výsledky sa nepoužijú a nemajú vplyv na systém. [3]

```
1. def priklad_irelevantny_kod(a: int):
2.         a = a + 5
3.         a = a ^ 0x75fea12
4.         a = a << 3
5.         a = a >> 3
6.         a = a ^ 0x75fea12
7.         a = a - 5
8.         return (a & 0xFF) ^ 0xFF
```

Obr. 17: Ukážka irelevantného kódu.

Nami zanalyzovaná obfuskačná technika s irelevantným kódom spadá priamo do tejto definície a v prípade, že by bola používaná vo väčšom množstve a značne komplikovala analýzu, mohli by sme použiť existujúce prostriedky zamerané na túto techniku.

3.4 Stack strings

Stack strings je známa obfuskačná technika predstavujúca reťazce, ktoré sa poskladajú až za behu tak, že po jednom sa písmená presunú na potrebné pozície v zásobníku. Dôsledkom je to, že takéto reťazce nie sú okamžite viditeľné v pamäti, teda ani štandardnými nástrojmi, ktoré vyhľadávajú sekvencie čitateľných znakov.

Napriek názvu sa táto technika dá a zvykne jednoducho aplikovať aj na inú pamäť, ako napríklad haldu.

Najjednoduchším spôsobom, ako sa takéto správanie dalo docieliť, bolo nahradenie štandardnej inicializácie reťazca v C `char c[] = "hello";` s `char c[] = { 'h', 'e', 'l', 'l', 'o', 0x00};`. [4] Aktuálne verzie bežných kompilátorov však vyhodnocujú tieto priradenia rovnako – dnes je už teda potrebné skladať reťazce znak za znakom, aby sme docielili obdobné správanie.

```
// SHT_PROGBITS  [0x404020
// ram:00404020-ram:0040402
//

__data_start
data_start
    ??          00h
    ??          00h
    ??          00h
    ??          00h

neobfuskovany

    ds          "hello"
```

```
 2 undefined8 main(void)
 3
 4 {
 5   obfuskovany[0] = 'h';
 6   obfuskovany[1] = 'e';
 7   obfuskovany[2] = 'l';
 8   obfuskovany[3] = 'l';
 9   obfuskovany[4] = 'o';
10   obfuskovany[5] = 0;
11   printf(neobfuskovany);
12   printf(obfuskovany);
13   return 0;
14 }
15
```

Obr. 18: Ukážka stack strings.

Nami zanalyzovaná obfuskácia reťazcov využíva túto techniku s kombináciou vyššie popísaných C funkcií a nastražených nesúvisiacich reťazcov. Tieto vylepšenia znemožňujú použitie bežných riešení na odhalenie stack strings. Tiež je dôležité spomenúť, že obfuskácia toku riadenia bola aplikovaná až po tejto technike a mala by byť teda vyriešená prednostne.

4 Deobfuskácia

V nasledujúcej kapitole popíšeme, ako sme sa s použitými obfuskačnými technikami vysporiadali. Výsledný kód je dostupný v nasledujúcom github repozitári: github.com/eset/stadeo.

Na začiatok ešte poznamenáme, že každý prístup k súborom na disku a k IDA API je vykonaný pomocou RPyC a IDAPython, ako bolo popísané vo východiskovej kapitole, ak nebude povedané inak. Vstupný súbor taktiež vždy získame pomocou metódy *idaapi.get_input_file_path()* zo vzdialenej inštancie Python-u s prístupom k IDA API.

4.1 Deobfuskácia vyrovnaného toku riadenia

Deobfuskáciu vyrovnaného toku riadenia sme sa rozhodli rozdeliť na tri samé o sebe dostatočne rozsiahle a nezávislé podkapitoly.

V prvej podkapitole sa zameriame na samotnú identifikáciu vyrovnaného toku riadenia, ktorá je nevyhnutná kvôli tomu, že sa v jedinej funkcii môže vyskytovať aj desiatky krát a manuálne rozoznávanie by bolo teda pracné. Žiadny z existujúcich spôsobov detekcie klasického vyrovnávania toku riadenia, ktorý sme videli, nebol použiteľný.

V druhej podkapitole popíšeme spôsob, ktorým sme detegovaný vyrovnaný tok riadenia deobfuskovali.

V tretej podkapitole popíšeme spôsob, ktorým automaticky deobfuskujeme väčšie časti programov. Naše riešenie nepokrýva funkcie, ktoré nie sú volané priamo, ale napríklad cez virtuálne tabuľky, čo je netriviálny problém mimo rozsah práce.

4.1.1 Identifikácia vyrovnaného toku riadenia vo funkcii

Pre účel identifikácie vyrovnaného toku riadenia vo funkcii sme vytvorili niekoľko tried, ktoré logicky rozdeľujú náš problém na menšie. Pri ich popise budeme postupovať od najvšeobecnejších k špecifickejším.

CFFRecognizer je hlavnou triedou a využíva priamo alebo nepriamo všetky ostatné. Obsahuje metódu *recognize()*, ktorá odštartuje algoritmus samotnej identifikácie.

Vyrovnávanie toku riadenia implementované skupinou Stantinko môže spájať podľa predchádzajúcej analýzy viaceré funkcie. Aká pôvodná funkcia bude vykonaná, je určené podľa *riadiacej premennej* v parametri výslednej funkcie počas behu. Z tohto dôvodu metóda *recognize()* akceptuje nepovinný parameter *merging_var_candidates*, ktorý obsahuje množinu premenných, ktoré môžu byť potenciálne takýmto parametrom.

Premenné triedy *CFFRecognizer* po identifikácii budú obsahovať všetky potrebné dáta pre deobfuskáciu funkcie aj na základe hodnoty *riadiacej premennej* v parametri, ak taká bude.

recognize() na začiatku disassembluje funkciu na danej adrese a adresy aktuálne patriace k blokom inštrukcií odstráni, pretože bloky budeme neskôr kopírovať. Bol by totiž problém, keby viacerým blokom prislúchala rovnaká adresa.

Následne preloží disassemblovaný kód do Miasm IR štandardným spôsobom a upraví ho tak, aby na kód mohol byť aplikovaný graf dátových závislostí, čo rieši v metóde *_normalize_ircfg()*. To znamená, že sa zaoberá niekoľkými problémami:

1. je potrebné zjednotiť aliasy v pamäti, ktorú budeme chcieť sledovať, teda v našom prípade smerníky na zásobník a jeho bázu. [17]
2. inštrukcie ako `mov bl, 1` spôsobujú problémy v prípade, že register `ebx` bol premennou, ktorú sme chceli sledovať. Nakoľko sa daná inštrukcia preloží ako `EBX = {1, EBX[8:]}` a k `bl` sa ďalej bude pristupovať ako k `EBX[:8]`, algoritmus dátových závislostí `bl` stále vníma ako závislé, aj keď vo skutočnosti už nie je, čo je potrebné vyriešiť.
3. série inštrukcií ako `test eax, eax; jnz …; lea edi, [eax+4]` spôsobujú problémy v prípade, že register `eax` bol premennou, ktorú sme chceli sledovať a zároveň má blok začínajúci inštrukciou `lea` iba jedného predchodcu. Je zrejmé, že `edi` bude obsahovať konštantu 4, ale algoritmus dátových závislostí ho stále vníma ako závislý, aj keď vo skutočnosti už nie je, čo je potrebné vyriešiť.

Prvý problém je riešený tak, že pri prechádzaní grafu funkcie do šírky sa pri každom prístupe k smerníku na zásobník získa z IDA API cez *idc.get_spd()* skutočná výška zásobníku v danom bode. Podľa tejto skutočnej výšky sa vypočíta adresa, ku ktorej skutočne pristupujeme [17]. Taktiež sa každý smerník na bázu zásobníku vyjadrí priamo cez smerník na zásobník, čo vo valnej väčšine prípadov vyriešilo náš problém. Zvyšné nevyriešené prípady boli knižničné funkcie.

Elegantnejším riešením by mohla byť kombinácia prevodu do SSA a späť do Miasm IR s propagáciou výrazov. Na viacerých testovaných prípadoch pre nás fungovala dobre, nanešťastie má však implementácia v Miasm-e momentálne niekoľko nevyriešených problémov. Tieto problémy sa prejavili pri pokusoch o deobfuskovanie viacerých funkcií. Keďže oprava týchto problémov je mimo rozsah práce, uprednostnili sme vyššie popísané riešenie.

Druhý problém je riešený tak, že na pravej strane problémových inštrukcií, ako je napríklad `EBX = {1, EBX[8:]}`, zanedbáme zvyšok pôvodnej premennej. Môžeme si to dovoliť, pretože takéto operácie nikdy neboli použité na *riadiacu*

premennú a meníme iba kód IR, čo sa neprejaví na výsledkoch. V uvedenom príklade vznikne teda `EBX = 1`.

V treťom probléme jednoducho na úrovni IR priamo povieme, že sledovaná premenná bude v danom bloku rovná nule za popísaných podmienok, algoritmus dátových závislostí ju potom už vynechá. Z `test eax, eax; jnz …; lea edi, [eax+4]` teda spravíme akoby `test eax, eax; jnz …; mov eax, 0; lea edi, [eax+4]`.

Na začiatok funkcie v takto modifikovanej IR ďalej ešte pridáme pre každý *merging_var_candidate* z *merging_var_candidates* priradenie v tvare *merging_var_candidate* = *merging_var_candidate*. Takýto výraz neskôr bude predstavovať jedinú výnimku, od čoho, okrem konštánt, môžu byť závislé *riadiace premenné*.

Na takto upravené IR cieľovej funkcie aplikujeme algoritmy analýzy kódu programov, ktoré budú potrebné neskôr. Ich výsledky a tento proces zastrešuje trieda *Analyses*, ktorá vypočíta konkrétne:

- Definition-Use chain pomocou triedy *Miasm.DiGraphDefUse*.
- Okamžitých dominátorov pomocou metódy *Miasm.compute_immediate_dominators()*.
- Dominátorov pomocou metódy *Miasm.compute_dominators()*.
- Spätné hrany, ktoré vypočítame podľa definície a už získaných dominátorov.

Vytvoríme zoznam všetkých funkcií *func_addresses* v programe cez IDA API pomocou *idautils.Functions()*.

Ďalej môžeme prejsť k samotnému algoritmu identifikácie. Prechádzame blokmi funkcie do hĺbky. Každý blok vykonáme symbolicky, pričom sa ako úvodný stav použije výsledný predchádzajúceho bloku a na začiatku prázdny.

Pozrieme sa na každú inštrukciu bloku a ak:

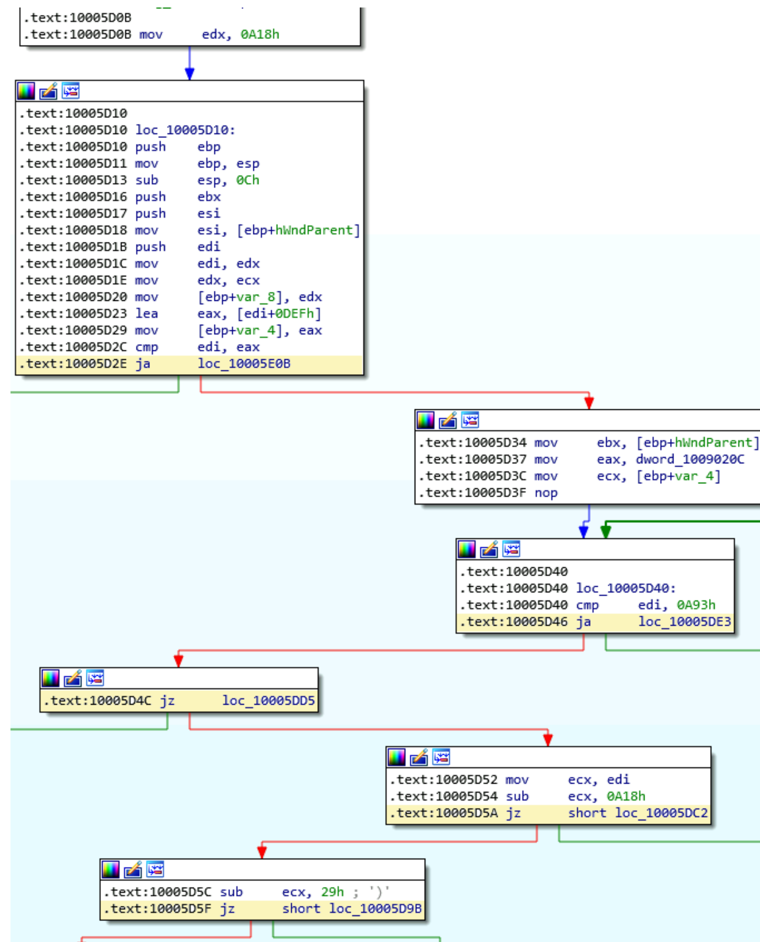
- Je výraz na pravej strane typu int, nachádza sa vo *func_addresses* a nepoužíva sa nikde vo funkcii, predpokladáme, že ide o časť virtuálnej tabuľky. Výraz zaevidujeme vtedy s pozíciou v množine *possible_merge_funcs*, pomocou ktorej

sa po deobfuskácii kontroluje dosiahnuteľnosť všetkých objavených funkcií. Dosiahnuteľné funkcie budú zaevidované a môžu byť deobfuskované neskôr.

- Je inštrukcia volanie a jej cieľ vo *func_addresses*, zaevidujeme jej cieľ s pozíciou v množine *possible_merge_funcs*. Evidujeme súčasne aj množinu *new_merging_var_candidates* obsahujúcu všetky výrazy z aktuálneho symbolického stavu, pričom pri týchto výrazoch musí byť ľavá strana pamäť alebo univerzálny register a pravá int. Pamäť závislá od zásobníku pritom bude opäť prepočítaná podľa skutočnej výšky zásobníku v danom bode.
- Ide o *merging_var_candidate = merging_var_candidate* alebo je výraz na ľavej strane univerzálny register alebo pamäť a výraz na pravej int, máme kandidáta na *riadiacu premennú*. Metóda *_process_assignment()* ďalej overí, či to tak skutočne je. Ak sa to potvrdí, *merging_var_candidates* sú vynulovaní, keďže už bol nájdený správny *merging_var*.

_process_assignment() na začiatku získa pomocou Definition-Use chain-u z *Analyses* všetky inštrukcie vetvenia *affected_irdsts* ovplyvnené daným priradením. Získa ich tak, že prechádza všetky použitia počnúc sledovaným priradením a zaznamená všetky nájdené inštrukcie vetvenia.

Na nasledujúcom obrázku sú zvýraznené inštrukcie vetvenia závislé od prvej inštrukcie, ktorá je priradením.



Obr. 19: Ukážka závislých inštrukcií vetvenia od priradenia.

Bloky funkcie sú následne prechádzané do šírky. Ak je práve prechádzaný blok *init_node* súčasťou *affected_irdsts*, počnúc ním sú *affected_irdsts* ďalej ešte filtrované. Sú filtrované tak, aby obsahovali iba sériu po sebe idúcich blokov, v ktorej bloky s viac ako jedným potokom musia byť súčasťou *affected_irdsts*.

Týmto chceme do istej miery vylúčiť vplyv vyššie popísanej anomálie číslo 5, ktorou sa budeme zaoberať ešte neskôr inými opatreniami, ktoré by mali eliminovať zvyšné nežiadúce prípady, ale pomalšie.

Ak sa niekde počas tohto procesu ukázalo, že nejaký z *affected_irdsts* je už súčasťou iného vyrovnaného toku riadenia, proces skončí neúspešne.

S pomocou takto odfiltrovaných *affected_irdsts*, ktoré budeme označovať ako *assign_blocks*, a *init_node* budeme pokračovať metódou *_create_flattenning_loop()*.

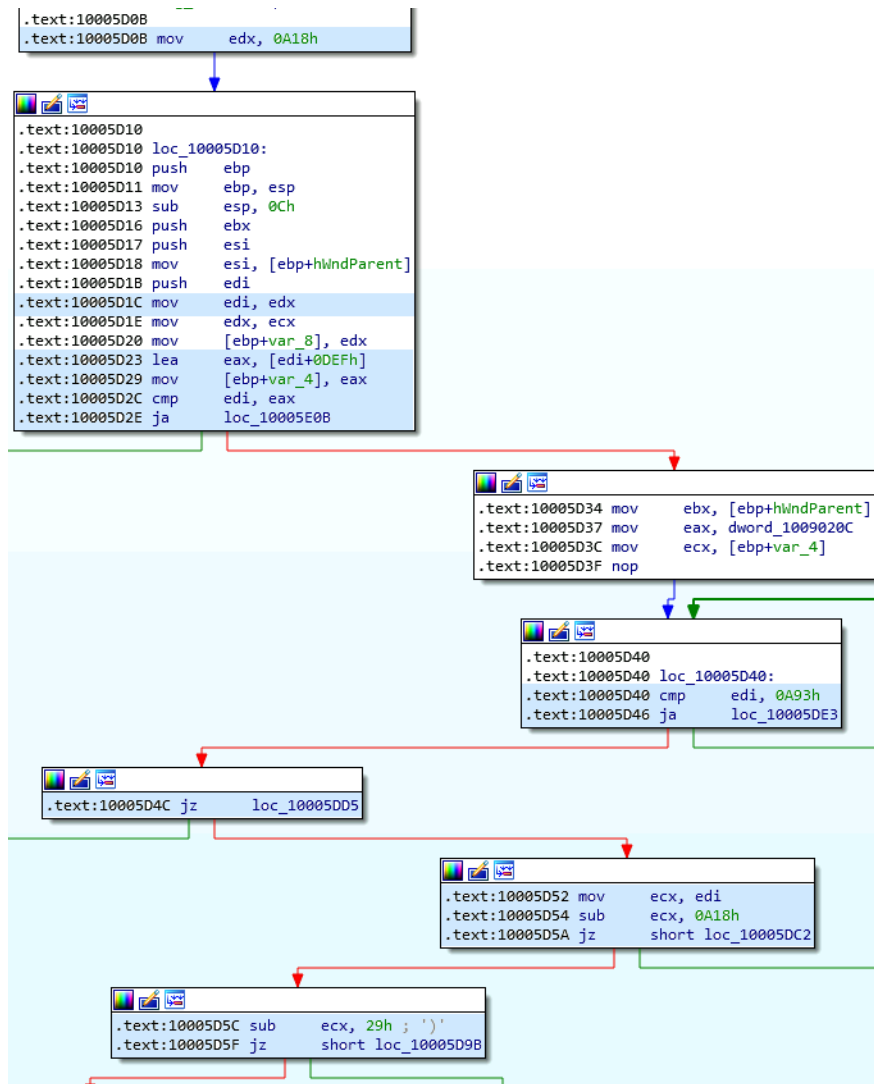
_create_flattenning_loop() volá 4 ďalšie metódy *_process_affected_irdsts()*, *_add_2_control_vars()*, *_cff_loop_sanity_check()* a *FlattenningLoops.create()*. Ak všetky uspejú, považujeme daný úsek kódu za vyrovnaný tok riadenia. Ďalej si ich popíšeme.

_process_affected_irdsts() vytvára graf dátových závislostí nad všetkými premennými na pravej strane každého priradenia z *assign_blocks* pomocou upravenej triedy *Miasm.DependencyGraph*. Úpravy tejto triedy neskôr ešte popíšeme. Zaujímajú nás iba korene tohto grafu, teda inštrukcie, ktoré sú závislé buď od konštánt, alebo premenných mimo našej funkcie.

V prípade, že je ľubovolný z koreňov závislý od premennej, ktorá nie je lokálna, považujeme ju za budúcu *merging_var*, čo pred priradením ešte overíme tak, že skontrolujeme, že:

- takáto premenná je iba jedna;
- ak už *merging_var* existuje, je to táto premenná;
- sa táto premenná nachádza medzi *merging_var_candidates*.

Ak táto kontrola neprebehla úspešne, dané priradenie vylúčime z *assign_blocks* a nepovažujeme za súčasť prípadného vyrovnaného toku riadenia, čím dôslednejšie eliminujeme vplyv anomálie číslo 5. V opačnom prípade ešte všetky závislé inštrukcie pridáme do množiny *affected_lines*.



Obr. 20: affected_lines vygenerované z predchádzajúcej ukážky.

Úpravy v *Miasm.DependencyGraph* majú za úlohu dve hlavné funkcie.

Prvá kontroluje, či spracovávaná premenná nie je parametrom nejakej volanej funkcie. Keďže Miasm nedokáže rozoznať parametre na zásobníku, nevidí sám takéto závislosti.

Pozrieme sa teda na najbližšie volania pri premenných, ktoré sú vkladané na zásobník. Následne pomocou *idaapi.get_arg_addrs(caller)* z IDA API skontrolujeme, či sa nejedná o parameter nejakej z následujúcich funkcií.

Ak sa skutočne jedná o parameter, je hneď jasné, že priradenie nemôže byť súčasťou vyrovnaného toku riadenia a algoritmus predčasne skončí chybou.

Pomocou Definition-Use chain z *Analyses* taktiež kontrolujeme následné použitia spracovávaných premenných, aby sme pokryli aj parametre, ktoré nie sú vkladané na zásobník ihneď, ale napríklad ako `lea eax, ...; push eax`.

Druhá úprava má za úlohu zrýchliť celý tento proces pomocou cache. Nakoľko pracujeme s veľkými funkciami a *assign_blocks* obsahuje značný počet priradení, tento proces býva časovo poznateľne náročný.

Očakávame, že všetky priradenia v *assign_blocks* idú za sebou a mali by byť závislé na rovnakých premenných. Na základe tohto predpokladu si k už spracovaným stavom poznačíme výsledok. Výsledok značí to, či sa skutočne jednalo alebo nejednalo o súčasť vyrovnaného toku riadenia. Počas chodu algoritmu potom kontrolujeme, či sme v takomto stave už neboli.

Kompletný graf dátových závislostí vytvoríme z legitímnych priradení v *assign_blocks* teda iba raz.

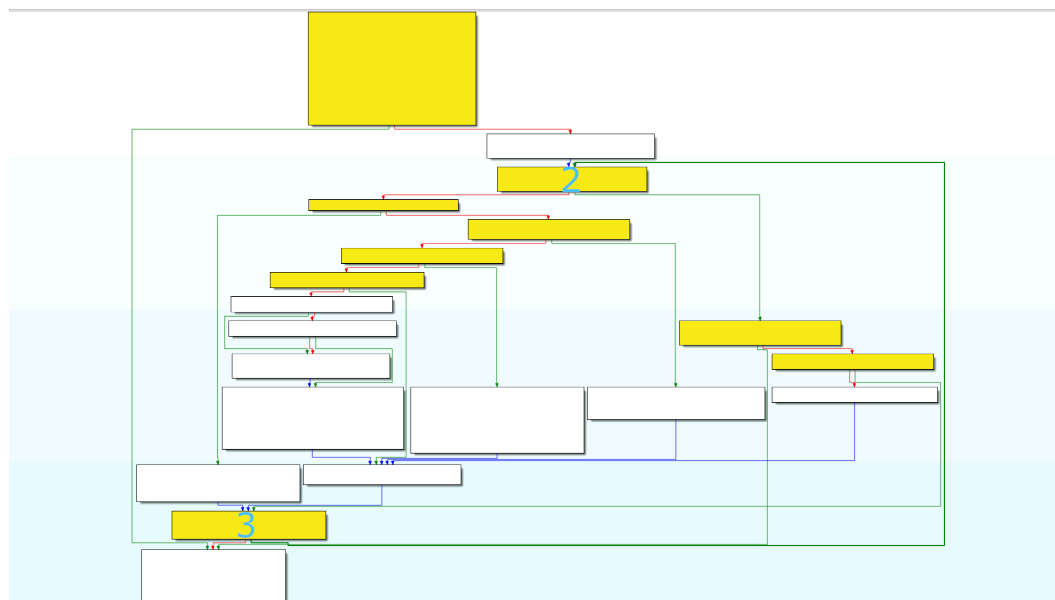
_add_2_control_vars() rieši primárne anomáliu číslo 8. Robí to tak, že prechádza všetky spätné hrany a hľadá takú, ktorá vedie do nejakého bloku z *assign_blocks*. Keď takú hranu nájde, pozrie sa, či má aspoň 2 potomkov a nenachádza sa v *affected_lines*.

Ak to nespĺňa, vezme predka, pretože hľadá druhú synchronizovanú *riadiacu premennú*, ktorá nemusela byť zvyšovaná v úplne poslednom bloku úseku – čo môže spôsobiť napríklad register spilling.

Ak takýto blok ešte nie je v *assign_blocks* a má aspoň 2 potomkov, skúsime ho pridať medzi *assign_blocks*. Zavoláme následne metódu *_process_affected_irdsts()* tak, aby skontrolovala iba práve pridaný blok. Ak metóda ďalej rozhodne, že nespĺňa vyššie popísané podmienky, tento blok jednoducho zmaže z *assign_blocks*.

Sekundárne tiež sleduje, či existuje aspoň jedna spätná hrana, ktorá spája 2 bloky z *assign_blocks*. Ak neexistuje, vyhodí chybu, lebo nemôže ísť o vyrovnaný tok riadenia. Dôvodom je to, že podľa analýzy vždy existuje aspoň 1 blok, *chvost*, ktorý vedie do *hlavy*, ktorá je vždy objavená skôr ako *chvost*.

Následujúci obrázok ilustruje spätnú hranu z bloku číslo 3 do bloku číslo 2 vo vyrovnanom toku riadenia. Žltou farbou sú zvýraznené bloky, ktoré sú súčasťou *assign_blocks*.



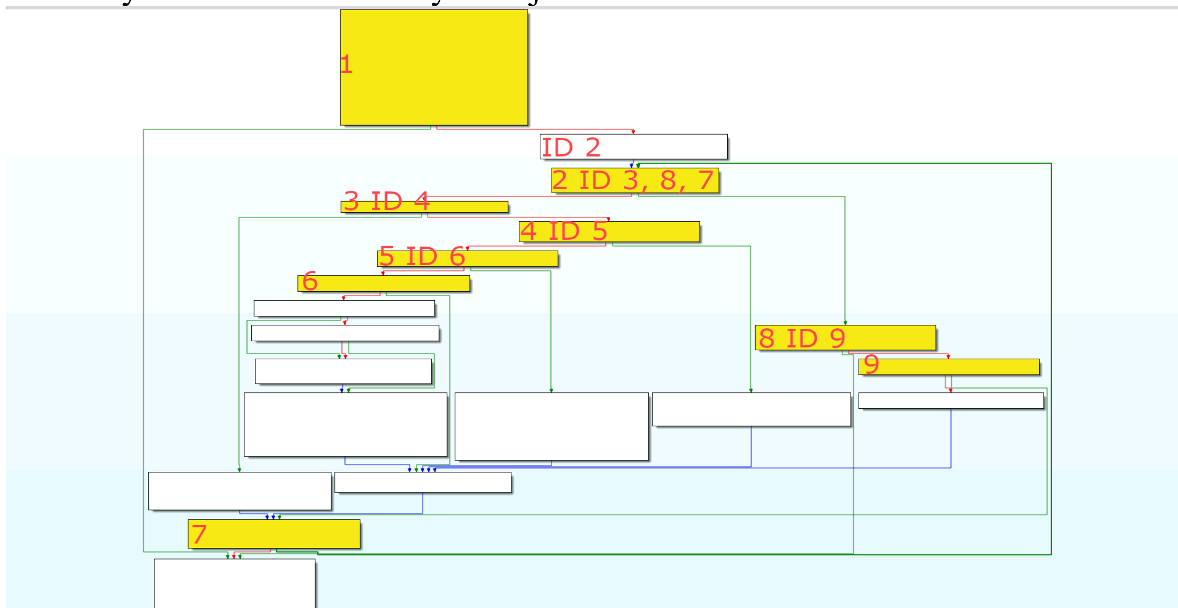
Obr. 21: Ilustrácia spätnej hrany pri vyrovnanom toku riadenia.

_cff_loop_sanity_check() kontroluje, či pre každý blok z *assign_blocks* platí, že ak daný blok nedominuje všetky ostatné bloky z *assign_blocks* – pri spojených funkciách býva na začiatku kontrola, či parameter spadá do určitého rozsahu – alebo nie je cieľom v spätnej hrane, teda *hlava*, tak okamžitý dominátor daného bloku – najbližší blok, cez ktorý treba prejsť, aby sme sa k nemu dostali – je v *affected_lines*,

Táto kontrola nastane iba v prípade, že aj priamy predchodcovia daného bloku sú v *affected_lines* alebo je sám súčasťou spätnej hrany. Toto overujeme preto, aby sme vylúčili určité optimalizácie kompilátoru – najmä register spilling – vytvárajúce ďalšie bloky s priradeniami medzi *assign_blocks*. Nakoľko sme pred tým už vybrali iba za sebou idúce bloky do *assign_blocks*, nezľahčuje to podmienky tak, ako sa môže na prvý pohľad zdať.

Tieto podmienky by sa dalo napísať ešte prísnejšie a sofistikovanejšie, ale aj tak by stále mohli vzniknúť problémové prípady. Nakoľko sa podmienky v tejto podobe veľmi dobre osvedčili, momentálne ich nechávame také, aké sú. V prípade, že sú porušené, prirodzene bude vyhodená výnimka.

Následujúci obrázok ilustruje okamžitých dominátorov ID (Immediate Dominator) vo vyrovnanom toku riadenia. Žltou farbou sú zvýraznené bloky, ktoré sú súčasťou *assign_blocks*. Číslo pred ID označuje blok a tie za ním čísla blokov, ktorých okamžitým dominátorom daný blok je.



Obr. 22: Ilustrácia okamžitých dominátorov vo vyrovnanom toku riadenia.

FlattenningLoops.create() zabezpečuje evidenciu novoobjaveného vyrovnaného toku riadenia v určitej funkcii. Informácie, ktoré sú potrebné k deobfuskácii, sú ukladané v inštanciách triedy *FlattenningLoop*. Tieto informácie obsahujú už popísané *affected_lines* a *assign_blocks*, ale aj množinu *head_vars*, v ktorej sú určité lokálne premenné. Od týchto premenných je závislý daný vyrovnaný tok riadenie a sú používané ako *riadiaca premenná*.

head_vars boli evidované ešte v *_process_affected_irdsts()*, ale boli zamlčané, keďže v danom kontexte neboli podstatné. Ich cieľom je sledovať také lokálne premenné, ktoré nie sú inicializované pred kódom vyrovnaného toku riadenia, ale až počas neho. Potom, ako sú premenné z *head_vars* nastavené, môže byť vykonaných niekoľko ďalších úsekov, kým sa podľa nich nastaví *riadiaca premenná*. S vysokou istotou si myslíme, že sú týmto spôsobom spracovávané jump tabuľky.

Ďalej je evidované asociatívne pole *affected_exprs*, ktoré mapuje k určitému bloku všetky premenné použité v *affected_lines* v rámci daného bloku. Hodnoty týchto premenných budú sledované pri deobfuskácii a na základe nich sa pri presúvaní blokov bude určovať, či už úsek s danou hodnotou riadiacej premennej deobfuskovaný bol. V

takom prípade budeme vedieť, že stačí vytvoriť skok na už deobfuskovaný úsek a netreba ho riešiť opakovane. Ak by sme ho však riešili opakovane, skončili by sme v nekonečnej slučke.

Mapovanie premenných ku konkrétnym blokom v *affected_exprs* robíme z toho dôvodu, že sa kvôli optimalizáciám kompilátoru používa pre *riadiacu premennú* viacero registrov a miest v pamäti, ktoré nechceme všetky naraz sledovať.

Dôvod je ten, že môže byť v jednom bloku napríklad použitý ako *riadiaca premenná* register `esi`. Mohla by nasledovať kvôli binárnemu vyhľadávaniu inštrukcia `lea edi, [esi-0x123]`.

Ďalší blok by už mohol používať `edi` ako riadiacu premennú a zároveň by kompilátor vedel, že všetky následujúce úseky budú používať napríklad náhodne vygenerované číslo.

Kompilátor by mohol usúdiť, že je vhodný čas vygenerovať ho v danom bloku a uložiť ho napríklad v už voľnom registri `esi`. Nemusel by ho potom generovať v každom z niekoľkých nasledujúcich úsekov samostatne. My by sme mali však problém, ak by sme `esi` stále sledovali, pretože by obsahovalo náhodné číslo, ostali by sme teda v nekonečnom cykle.

Trieda *FlattenningLoop* taktiež obsahuje jedinú metódu *get_affected_hash()*, ktorej úlohou je generovanie jedinečných identifikátorov blokov na základe ich príslušnosti k určitému vyrovnanému toku riadenia. Tieto bloky musia byť z *assign_blocks* a zároveň používať priamo hodnotu *riadiacej premennej*.

Pod priamym použitím hodnoty *riadiacej premennej* myslíme to, že napríklad v prípade sledu inštrukcií `cmp eax, 0x123; ja …; jz …`, v ktorom je register `eax` riadiaca premenná, je inštrukcia `jz` v samostatnom bloku. To, čo ostalo z riadiacej premennej v tomto bloku, je hodnota nulového príznaku 0 alebo 1. Z tohto rozhodne nedokážeme určiť, či jeden z napríklad desiatich nasledujúcich úsekov bol už deobfuskovaný.

Jedinečný identifikátor je vypočítaný ako hash z hodnôt *head_vars* a *affected_exprs* patriacich danému bloku. Hodnoty sú získané poskytnutým symbolickým stavom. Bloky, ktoré nie sú v *affected_exprs*, majú priradený vždy náhodný hash z vyššie popísaného dôvodu. Predpokladá sa, že všetky bloky, s ktorými bude pracovať, obsahujú inštrukcie z *affected_lines*.

V tomto bode je dobré poznamenať, že nakoľko *head_vars* vplývajú na každý hash cez viaceré úseky, v takýchto prípadoch budú momentálne vznikať duplicity. Toto sa v praxi neukazuje byť veľký problém, keďže sa s nimi moderné dekompilátory dokážu dostatočne dobre popasovať. Takéto prípady sa taktiež vyskytli veľmi zriedka, čo bol hlavný dôvod, prečo sme nezvažovali ďalšie kroky.

FlattenningLoop.get_affected_hash() používame výhradne cez *FlatteningLoops.get_block()*, ale pred tým, ako sa k nemu dostaneme, potrebujeme popísať triedu *FlatteningBlock*. Táto trieda jednoducho rozšíri bežný identifikátor základných blokov predstavujúci buď štandardnú adresu, alebo formu ID v Miasm-e. Tento identifikátor bude generovať ako hash z nasledujúcich prvkov, z ktorých niektoré môžu byť aj None:

- ID bloku v Miasm-e *block_loc_key*;
- ID *loop_loc_key* prislúchajúce inštancii triedy *FlattenningLoop*;
- Hash *control_hash_value* vygenerovaný pomocou *FlattenningLoop.get_affected_hash()*;
- ID inštancie triedy *FlattenningLoop* prislúchajúcej k predchádzajúcemu bloku *source_loop_loc_key*;
- Hash predchádzajúceho bloku *source_hash_value*.

FlatteningLoops.get_block() vytvára inštanciu triedy *FlatteningBlock* pomocou poskytnutého *block_loc_key*, aktuálneho symbolického stavu a voliteľne *source_flat_block*, teda predchádzajúcej inštancie *FlatteningBlock*.

Prvým krokom je kontrola, či je daný blok súčasťou nejakého známeho vyrovnaného toku riadenia. V takom prípade sa vypočíta štandardne hash *control_hash_value*. *source_hash_value* so *source_loop_loc_key* sú nastavené na None, nakoľko nás tieto hodnoty zaujímajú iba pri blokoch, ktoré nie sú súčasťou žiadneho vyrovnaného toku riadenia.

Ak daný blok nie je súčasťou žiadneho vyrovnaného toku riadenia, skontrolujeme, či je súčasťou *affected_lines*. Ak je, vypočítame štandardne hash *control_hash_value*, *source_hash_value* nastavíme na None a *source_loop_loc_key* na *loop_loc_key* predchádzajúceho bloku *source_flat_block*. Týmto ho považujeme za akúsi polovičnú súčasť vyrovnaného toku riadenia – žiadny jeho potomok nebude zmazaný a ani vymenený za iný blok pri samotnej deobfuskácii. No nakoľko prišlo k

zmene hodnoty riadiacej premennej, musíme prirodzene zmeniť príslušný hash, aby sme nepovažovali dve rôzne hodnoty riadiacej premennej za jednu.

Ďalší prípad je, keď daný blok nie je súčasťou žiadneho vyrovnaného toku riadenia a ani súčasťou *affected_lines*. V takejto situácii nastavíme *control_hash_value* na None, *source_loop_loc_key* a *source_hash_value* podľa hodnôt predchádzajúceho bloku *source_flat_block*.

Posledný neopísaný prípad je taký, keď blok nie je súčasťou žiadneho vyrovnaného toku riadenia a *source_flat_block* nebol poskytnutý. Vtedy nastavíme jednoducho všetko okrem *block_loc_key* na None, keďže sme ešte žiadny vyrovnaný tok riadenia neobjavili a nemusíme nič riešiť.

Opäť poznamenáme, že existujú prípady, v ktorých vznikajú duplicity, ktoré tiež efektívne riešia dekompilátory. Ak konkrétne narazíme na štandardný blok, ktorý sme už videli, ale s iným *source_loop_loc_key*, znamená to, že jedna z ciest viedla k danému bloku cez iný vyrovnaný tok riadenia a blok sa bude duplikovať. Vo skutočnosti sa duplikovať však nemusí.

Tieto duplicity by bolo možné eliminovať napríklad tak, že by každý *FlatteningBlock* neevidoval jediný pár *source_loop_loc_key* s hashom predchádzajúceho bloku, ale pre každý posledný *FlattenningLoop* voliteľne jeden.

Na záver podkapitoly ešte spomenieme, že počet nesprávne identifikovaných vyrovnaných tokov riadenia bol zanedbateľný. Navyše zatiaľ nebol prípad, ktorý by vygeneroval nesprávny kód, vždy akurát prišlo k rozvinutiu určitej slučky spĺňajúcej podmienky.

Takéto rozvinutie môže spraviť kód menej prehľadný, ale keďže je to na prvý pohľad viditeľné, môžeme nesprávne identifikovaný vyrovnaný tok riadenia manuálne vylúčiť a deobfsukovať danú funkciu opätovne bez neho.

Tieto prípady nie je možné s určitosťou eliminovať, ale bolo by možné pridať heuristiku, ktorá ich ešte ďalej bude filtrovať. Nakoľko bolo takýchto prípadov veľmi málo, ďalej sme sa tým nezaoberali.

4.1.2 Deobfuskácia identifikovaného vyrovnaného toku riadenia vo funkcii

Samotná deobfuskácia je po identifikácii oveľa menší problém. Samá o sebe obsahuje už iba jednu triedu *CFFSolver*. Využívajú sa však tie, ktoré sme vytvorili pri identifikácii. Pri inicializácii táto trieda očakáva inštanciu triedy *CFFRecognizer*.

Jej hlavnou metódou je metóda *process()*, ktorá akceptuje tri parametre:

- *reached_funcs*, ktorý predstavuje množinu objavených funkcií;
- *pending* predstavujúci asociatívne pole, ktoré priraďuje k adresám objavených funkcií už popísané množiny *CFFRecognizer.merging_var_candidates*;
- *merging_val*, ktorý predstavuje aktuálnu hodnotu už popísanej premennej *CFFRecognizer.merging_var*.

V prípade, že k danej funkcii, ktorú sa snažíme deobfuskovať nebol nájdený žiadny vyrovnaný tok riadenia, parametre *reached_funcs* a *pending* sú naplnené so všetkými príslušnými hodnotami z *CFFRecognizer.possible_merge_funcs* a funkcia skončí.

V prípade, že sú nastavené premenné *merging_val* a aj *CFFRecognizer.merging_var* na korektné hodnoty, na začiatok danej funkcie pridáme priradenie *merging_val* do *CFFRecognizer.merging_var*.

Pokračujeme volaním metódy *_deobfuscate_cff_loops()*, ktorá obsahuje už samotný deobfuskačný algoritmus.

Nakoniec do *reached_funcs* a *pending* priradíme opäť príslušné hodnoty z *CFFRecognizer.possible_merge_funcs*. No tento krát iba také, pri ktorých sme adresu volania danej funkcie aspoň raz videli pri deobfuskácii. Volania sa teda nachádzajú v *relevant_nodes* a sú určite súčasťou aj danej pôvodnej funkcie.

_deobfuscate_cff_loops() vytvorí nový symbolický stav a z prvého bloku danej funkcie získa inštanciu *FlatteningBlock* pomocou metódy *FlatteningLoops.get_block()*.

Všetky vytvorené inštancie *FlatteningBlock* sú pridané do asociatívneho poľa *flat_block_to_loc_key*, ktoré ku každej priraďuje nový Miasm identifikátor. Toto pole neskôr použijeme pre assemblovanie deobfuskovanej funkcie. Tiež z neho vieme povedať, ktoré inštancie *FlatteningBlock* už boli spracované.

Ďalej sa v slučke pracuje s každou ešte nespracovanou inštanciou *FlatteningBlock*, ku ktorej prislúcha určitý sybolický stav. Na začiatku je známa iba už popísaná prvá inštancia *FlatteningBlock*.

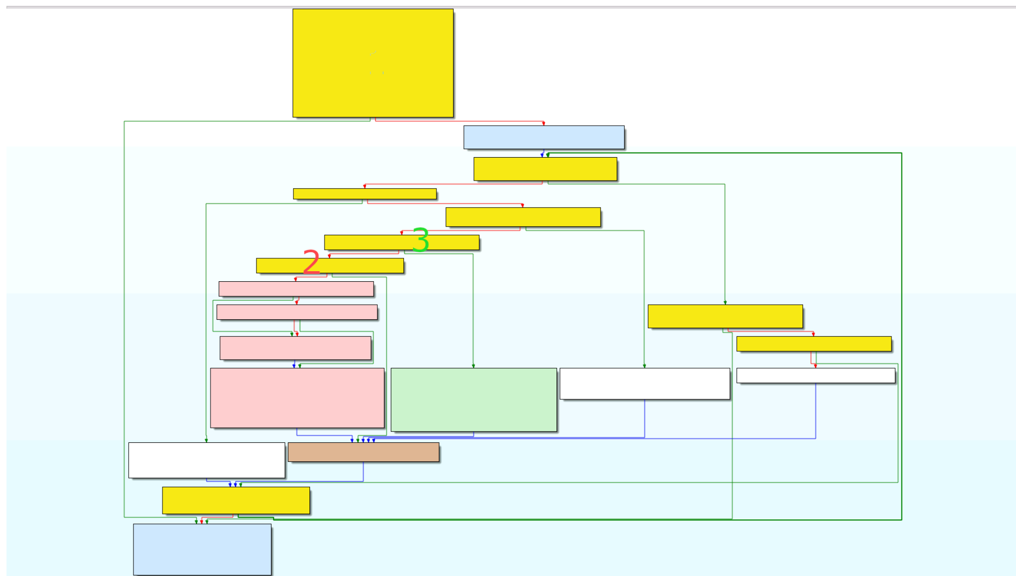
V slučke sa konkrétne zaradí príslušný *FlatteningBlock.block_loc_key* medzi *relevant_nodes* a symbolický stav sa aktualizuje podľa *affected_lines* v danom bloku.

Nakoniec sa daný blok, ako source_flat_block, vloží do metódy *_insert_flat_block()*, ktorá podľa inštancií *FlatteningBlock* postupne skladá nový graf. Metóda vracia novovytvorené inštancie *FlatteningBlock*, ktoré predstavujú nových potomkov a sú následne obdobne spracované.

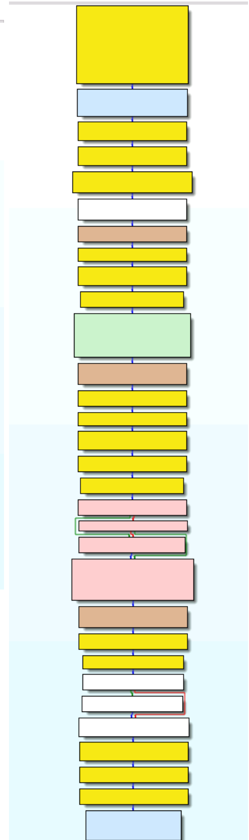
_insert_flat_block() vytvorí kópiu assembly bloku *new_block* z bloku, ktorému prislúcha Miasm ID *source_flat_block.block_loc_key*. Ak sa daný blok nachádza v *assign_blocks* určitej *FlattenningLoop*, vyhodnotí cieľovú symbolickú destináciu. Ak je destinácia jednoznačná, spraví na konci bloku nepodmienený skok smerujúci na ňu, inak vypíše varovanie, že niekde nastala chyba, pretože taká situácia by nemala nastať.

Nakoniec získa inštanciu *FlatteninnigBlock* pre každého potomka pomocou *FlatteningLoops.get_block()*, ak takýto blok ešte nebol videný, vloží ho do množiny, ktorú vracia. Neskôr budú obdobne spracovaní.

Následujúce obrázky ilustrujú, akoby mohla byť určitá funkcia deobfuskovaná. Je pritom vidno, že viaceré úseky, napríklad tie označené číslami 2 a 3, sa môžu prekrývať v hnedom bloku.



Obr. 24: Ilustrácia obfuskovanej funkcie.



Obr. 23: Ilustrácia deobfuskovanej funkcie.

4.1.3 Automatická deobfuskácia vyrovnávania toku riadenia v programe

Hlavnou triedou, ktorá zastrešuje kompletnú deobfuskáciu, je trieda *CFFStrategies*.

Postupne opíšeme všetky jej štyri metódy:

_solve_loop() je metóda, pri ktorej sa očakáva, že bude volaná iba prostredníctvom ostatných metód.

V parametroch dostáva adresu *empty_address*, na ktorú má byť vložená deobfuskovaná funkcia, inštanciu triedy *CFFRecognizer*, ktorá už spracovala určitú funkciu, a voliteľne hodnotu *merging_var*.

Následne inicializuje triedu *CFFSolver*, ktorej posunie inštanciu *CFFRecognizer* z parametra a zavolá jej metódu *process()* s hodnotou *merging_var*, ktorá je tiež z parametra. *pending* a *reached_funcs* sú pritom viazané na premenné triedy *CFFStrategies _pending* a *_reached_funcs*, ktoré sú inicializované na prázdny slovník a množinu.

Ďalej je deobfuskovaná funkcia zassemblovaná tak, ako bolo popísané vo východiskovej kapitole – zapíše sa na adresu *empty_address*. Nakoniec sa cez IDA API zavolá *idaapi.reload_file()* a *ida_funcs.add_func()*, aby sme novú funkciu aj evidovali.

Funkcia vracia *empty_address* zvýšenú o veľkosť výslednej deobfuskovanej funkcie. Ak nebude povedané inak, vždy sa na túto novú adresu nastaví predchádzajúca *empty_address*, ak sa spracuje viacero funkcií za sebou.

process_merging() je metóda, ktorá sa snaží počnúc určitou funkciou deobfuskovať všetky, ktoré sú dosiahnuteľné.

Akceptuje parametre:

- *func_addresses* – aktualizuje ním *_reached_funcs*.
- *empty_address* – adresa, na ktorej začína assemblovať všetky objavené deobfuskované funkcie.
- *recognized_funcs* – dobrovoľný parameter, ktorý môže byť výsledok predchádzajúceho behu tejto metódy. Preskakujú sa jeho pomocou už spracované funkcie.

Kým začneme so samotným opisom metódy, podotkneme, že keďže funkcie bývajú obrovské a proces identifikácie a aj disassemblovania časovo citeľne náročný, chceme sa vyhnúť opakovaniu týchto procesov. V prípade, že narazíme na už deobfuskovanú funkciu, ale s inou hodnotou *riadiacej premennej* v parametri, chceme pracovať s už disassemblovaným kódom a identifikovanými obfuskovanými časťami.

Pre tieto účely uchovávame určitý počet disassemblovaných funkcií v asociatívnom poli *recognized_funcs*, na ktoré bol aplikovaný algoritmus identifikácie vyrovnaného toku riadenia podľa maximálneho počtu disassemblovaných blokov. Po

prekročení daného limitu celkového počtu blokov, všetky takéto uložené dáta jednoducho zmažeme. V budúcnosti by mohli byť aj uložené na disk a odtiaľ v prípade potreby obnovené.

Metóda prechádza všetky adresy funkcií *func_addr* v *_reached_funcs*. Ak bol prekročený spomínaný maximálny počet disassemblovaných blokov všetkých funkcií v popísanej cache, v *recognized_funcs* budú vyčistené dané dáta.

Ak sa *func_addr* nenachádza v *recognized_funcs*, tak pomocou IDA API v prvom rade skontrolujeme, či nie je označovaná ako knižničná pomocou *idaapi.FUNC_LIB*. Ak tak označovaná je, preskočíme ju, inak ju spracujeme pomocou triedy *CFFRecognizer* a jej metódy *recognize()*. Následne danú inštanciu *CFFRecognizer* uložíme v *recognized_funcs* spolu s prázdnym asociatívnym poľom *vals*, ktoré mapuje hodnoty *merging_var* na adresy už deobfuskovaných funkcií.

Ak sa daná *func_addr* nachádza v *recognized_funcs*, ale nie v *_pending* alebo neobsahuje *merging_var*, tak sa nepokračuje ďalej. Spracuje sa vtedy ďalšia *func_addr*. V takom prípade buď nemôže mať viacero deobfuskovaných podôb, alebo sme už všetky známe vyriešili.

Ak bola príslušná inštancia *CFFRecognizer* vyčistená v rámci mazania cache, znovu voláme *recognize()*.

Ak k danej funkcii nebol nájdený *merging_var*, aplikuje sa na ňu metóda *solve_loop()*, adresa deobfuskovanej funkcie sa uloží v *recognized_funcs* a pokračujeme s ďalšou *func_addr*.

Ak *merging_var* nie je v *merging_var_candidates* pri danej adrese v *_pending*, znamená to, že pre toto volanie funkcie nebola nájdená vhodná hodnota daného parametra predstavujúceho *merging_var*. Pred tým mohol byť identifikovaný nesprávne. Vypíšeme v takom prípade varovanie, vynulujeme *merging_var* a pokračujeme s ďalšou *func_addr*.

Nakoniec prechádzame všetky ešte nespracované hodnoty, ktoré sme zatiaľ zistili, že môže *merging_var* nadobudnúť. Sú spracovávané pomocou metódy *_solve_loop()*.

solve_loop() je metóda, ktorá sa pokúsi deobfuskovať jednu konkrétnu funkciu. Akceptuje parametre:

- *func_address* – adresa funkcie, ktorá má byť deobfuskovaná.
- *empty_address* – adresa, kde má byť deobfuskovaná funkcia uložená;
- *context* – asociatívne pole, ktoré bude použité ako už popísané *merging_var_candidates*, pričom sa očakáva, že bude obsahovať práve jednu hodnotu.

Použije na to štandardne triedu *CFFRecognizer*, jej metódu *recognize()* a metódu *_solve_loop()*.
process_all() je metóda, ktorá prechádza všetky funkcie, ktoré sú v programe a predpokladá, že žiadna neobsahuje *merging_var*. Adresy funkcií sú získané pomocou metódy *idautils.Functions()* z IDA API.

Pre každú funkciu robí to, čo metóda *solve_loop()*, ale nepoužíva *context*, všetky deobfuskované funkcie sú postupne ukladané na *empty_address*.

4.2 Deobfuskácia reťazcov

Pre deobfuskáciu reťazcov použijeme primárne triedu *DSEEngine* z Miasm-u, ktorá bude pripojená k Sandbox-u požadovanej architektúry. Zameriame sa ďalej na to, ako pomocou nej vyhľadať takéto reťazce v konkrétnych funkciách.

Hlavnou myšlienkou je vyhľadať v pamäti, ktorá bola upravená počas behu funkcie, vytvorené reťazce. Nakoľko na to musia byť modifikované rozsahy pamäti symbolizované a z popisu obfuskačnej techniky vieme, že sa používajú C funkcie pri skladaní reťazcov, musíme registrovať príslušné symbolické funkcie pre každú takúto externe volanú funkciu. Také sa však ukázali byť iba dve, a to *lstrcpy()* a *lstrcat()*.

Prvým krokom samotného algoritmu bude inicializácia zásobníku, ktorému dáme dosť priestoru na to, aby následné prípadné pokusy poskladať na ňom reťazec, neskončili výnimkami. Pomocou metódy *DSEEngine.update_state_from_concrete()* zabezpečíme, že symbolický stav bude úplne konkrétny, nakoľko nás tieto zmeny nebudú zaujímať.

Pred tým, ako budeme pokračovať, aktuálny stav uložíme metódou *DSEEngine.take_snapshot()*, pretože sa k nemu budeme neskôr vracať. Aby sme neskôr nepadli do nekonečných slučiek, nastavíme taktiež spätné volanie, ktoré počíta,

koľkokrát bola každá inštrukcia spustená. Ak je dosiahnutá hodnota 500, prerušíme vykonávanie.

Následne spúšťame dynamické symbolické vykonávanie tak, aby sme vykonali všetky bloky funkcie, pričom sa pokúšame zájsť čo najďalej. V prípade, že určitá inštrukcia spôsobí výnimku prvý krát, preskočíme ju a pokúšame sa pokračovať. Ošetrujeme tak prípady, v ktorých môže kód v prostriedku skladania reťazca z inej príčiny trebárs pristúpiť k nedefinovanej pamäti, ktorou môže byť napríklad smerník z parametrov funkcie.

Takéto vykonávanie, ktoré začalo určitým blokom, beží pokiaľ nejaká inštrukcia nespôsobí výnimku druhý krát, nevykoná sa päťstý krát alebo sa nenarazí na koniec funkcie.

Na konci vykonávania v súvislých úsekoch upravenej pamäte vyhľadávame sekvencie čitateľných znakov pomocou regex-ov, ktoré považujeme za poskladané reťazce.

Ak ešte neboli vykonané všetky bloky, obnovíme uložený počiatočný stav pomocou metódy *DSEEngine.restore_snapshot()* a pokračujeme ďalším blokom.

V poslednom kroku vyradíme kratšie reťazce, ktoré sú obsiahnuté vo väčších – vo väčšine prípadov neboli ešte kompletné.

Túto metóda ďalej už len jednoducho rozšírime tak, aby mohla byť aplikovaná buď na množinu vybraných funkcií, alebo všetkých, ktoré získame pomocou funkcie *idautils.Functions()* z IDA API.

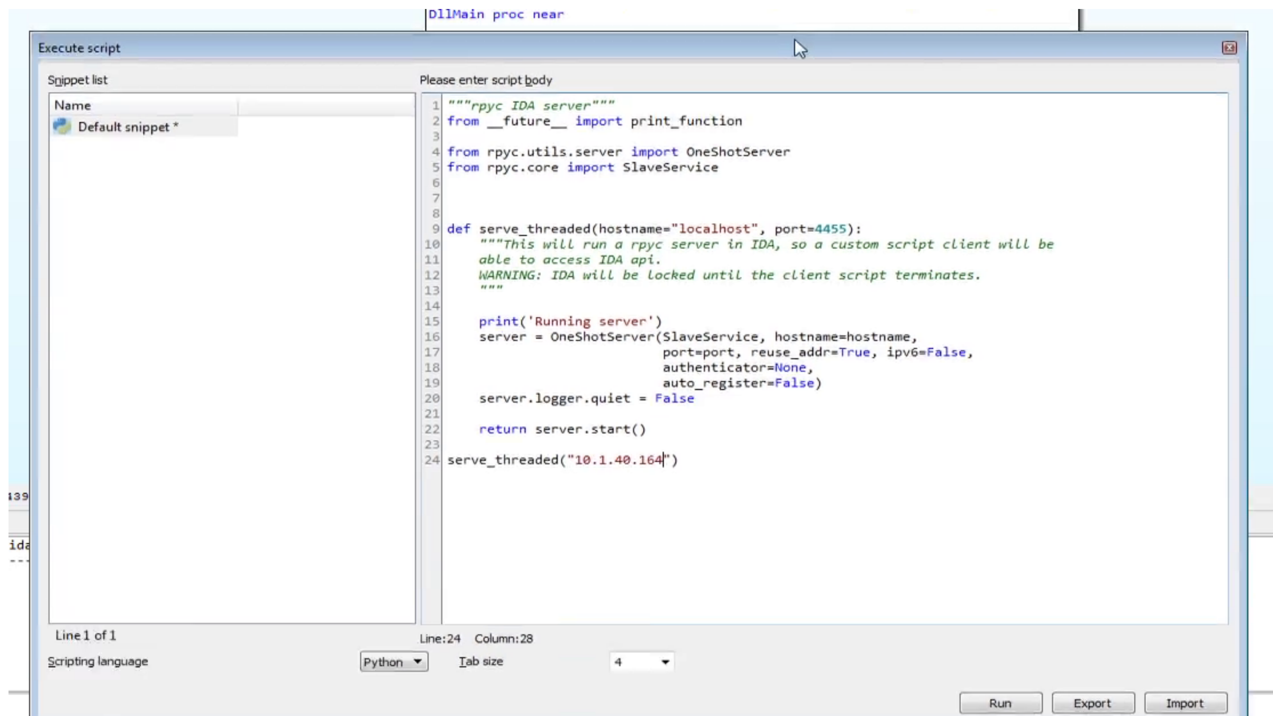
5 Výsledky

V nasledujúcej kapitole si ukážeme na niekoľkých príkladoch, ako naše riešenie vyzerá v praxi.

Kým začneme, ešte dodáme, že program je možné nainštalovať jednoducho pomocou príkazu `pip install git+https://github.com/eset/stadeo`. Pritom musia byť splnené všetky softvérové požiadavky Miasm-u popísané v github.com/cea-sec/miasm, ktoré automaticky nie sú inštalované, čo v našom prípade

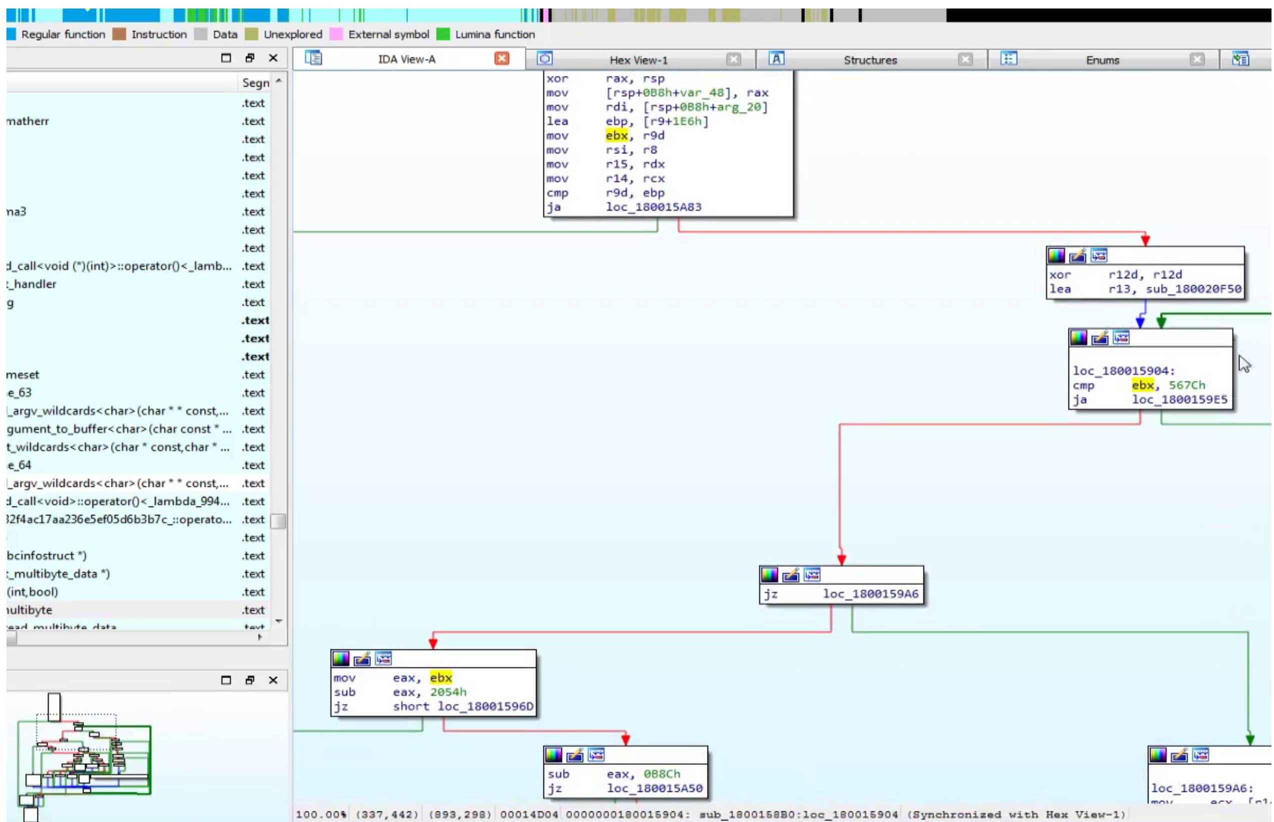
bola akurát inštalácia balíku python-dev. Program sme testovali iba pod operačným systémom Linux.

Tiež je potrebné podotknúť, že nakoľko sa pripájame k inštancii IDAPython-u cez RPyC, vždy je v nej potrebné spustiť RPyC server podobne ako na nasledujúcom obrázku. Daný skript je súčasťou Miasm-u a dostupný na github.com/cea-sec/miasm/blob/master/example/ida/rpyc_ida.py.

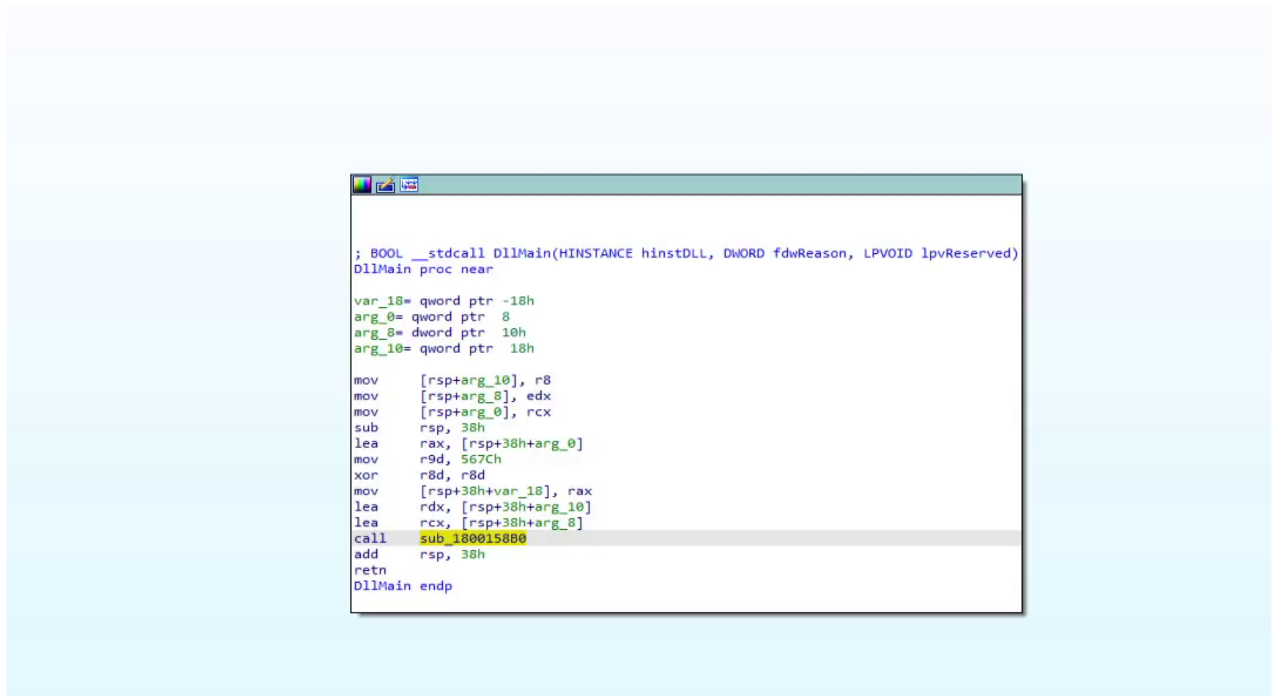


Obr. 25: Spustenie RPyC servera v IDE.

V prvej ukážke deobfuskujeme funkciu, ktorá spája niekoľko ďalších už spomínaným spôsobom pomocou parametra v registri `R9`.

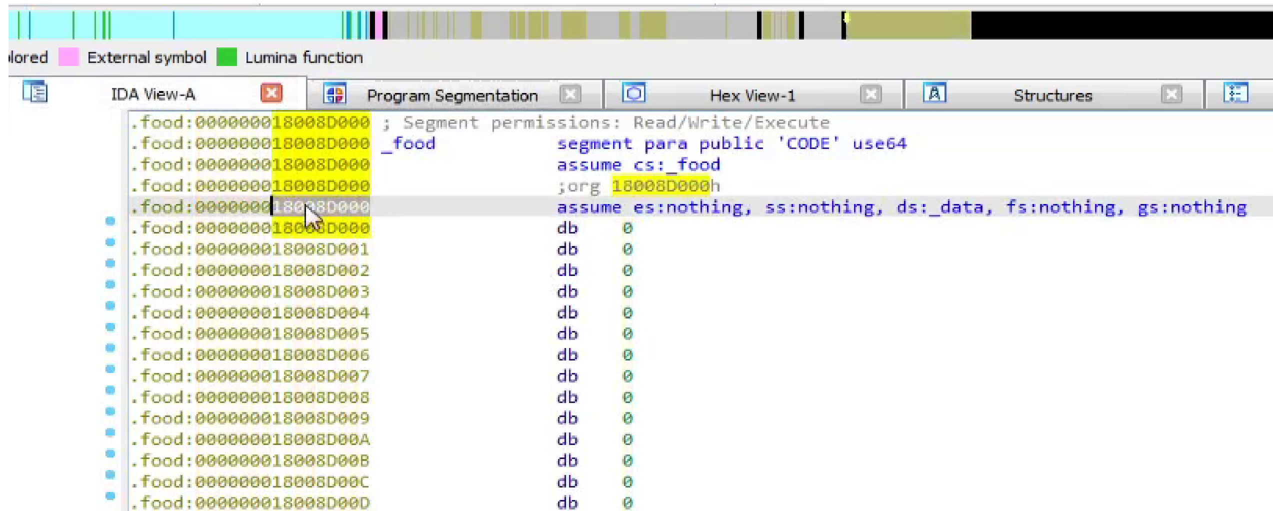


Obr. 26: Náhľad obfuskovanej funkcie 1.



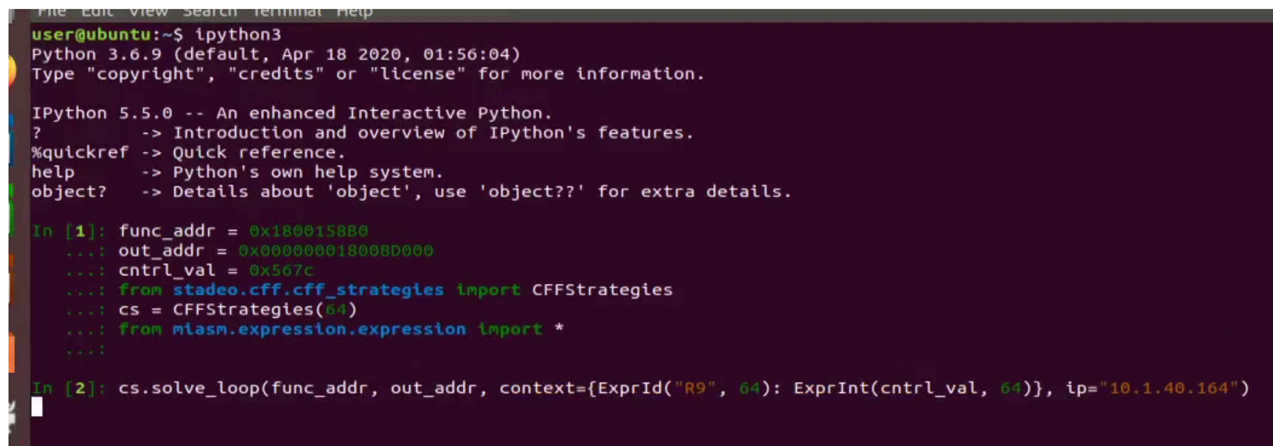
Obr. 27: Volanie obfuskovanej funkcie 1.

Ešte podotkneme, že pre deobfuskované funkcie potrebujeme nejaké miesto. Na tento účel sme vytvorili v súbore novú prázdnu sekciu.



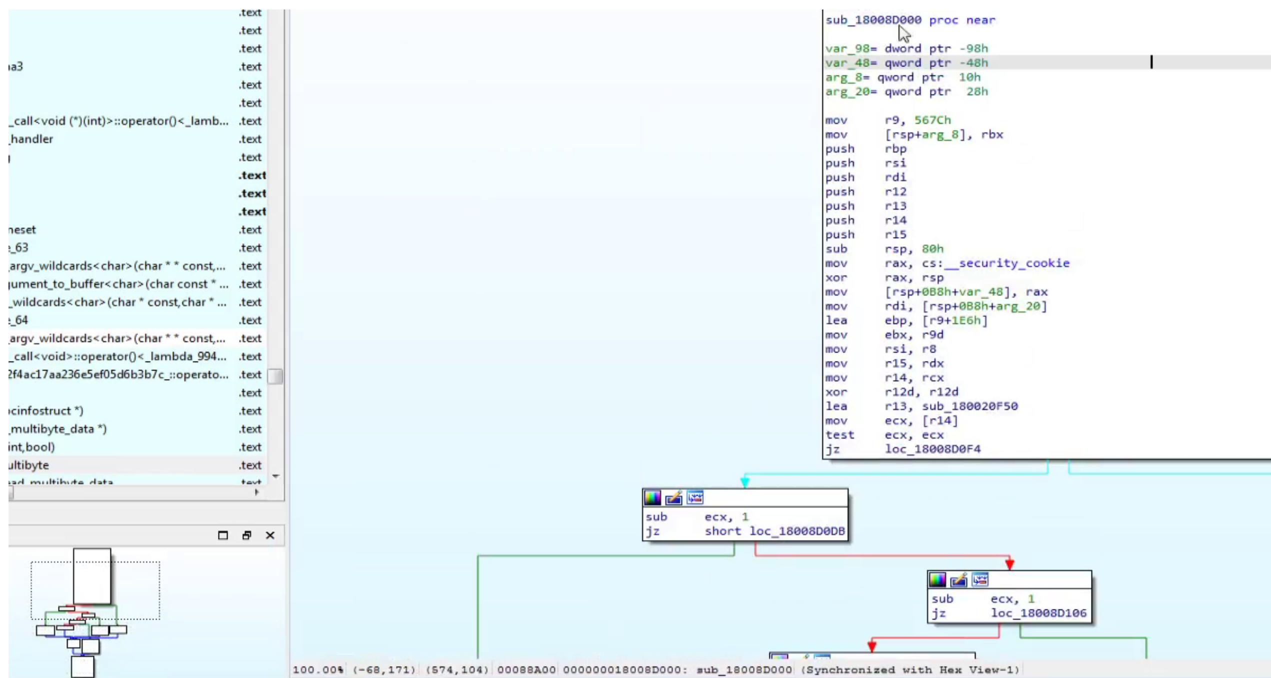
Obr. 28: Prázdna sekcia pre deobfuskovaný kód.

Následujúcimi príkazmi sa pripojíme k IDE a deobfuskujeme danú funkciu podľa príslušnej hodnoty v parametri `R9`, ktorý predstavuje *riadiacu premennú*. Deobfuskovaná funkcia bude vložená na začiatok vytvorenej sekcie.



Obr. 29: Kód pre deobfuskáciu funkcie 1.

Na ďalšom obrázku máme náhľad deobfuskovanej funkcie.



Obr. 30: Náhľad deobfuskovanej funkcie 1.

V druhej ukážke predvedieme kód, ktorý od určitej funkcie prechádza obfuskovaný program, vyhľadáva dosiahnuteľné obfuskované funkcie, postupne ich deobfuskuje a vkladá postupne do prázdnej sekcie.

Konkrétne vidíme, že objavil a deobfuskoval tri funkcie, a to na adresách `0x1001ffc0`, `0x1003f410` a `0x1000f0c0`. Tie boli následne deobfuskované vložené na adresy `0x10098000`, `0x100982cc` a `0x10098589`. Taktiež vieme vyčítať, že riadiace premenné boli v prvom a druhom parametri na zásobníku a ich hodnoty `0x6cef`, `0x21a4` a `0x2012`. Hodnoty riadiacich premenných a súčasti žiadnych deobfuskovaných vyrovnaných tokov riadenia, ktoré nespájajú viaceré funkcie, nie sú evidované.

Podotýkame, že kód by mohol bežať ďalej a spracovať viac funkcií, ale pre ilustračné účely nám bude stačiť iba pár prejdených funkcií.

Taktiež je vidno, že viaceré funkcie boli preskočené, buď z dôvodu, že boli označené IDOU ako knižničné, alebo ich náš algoritmus neoznačil ako obfuskované.

```
user@ubuntu:~$ ipython3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import rpyc
   ...: from stadeo.cff.cff_strategies import CFFStrategies
   ...: cs = CFFStrategies(32)
   ...: out_addr = 0x10098000
   ...: func_addr = 0x1002DC50
   ...:

In [2]: conn = rpyc.classic.connect("10.1.40.164", 4455)

In [3]: cs.process_merging(func_addr, out_addr, conn=conn)
skipping 0x1002dc50 with val None: 0xbadf00d
mapping: 0x1001ffc0 -> 0x10098000 with val @32[ESP_init + 0x8]: 0x6cef
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A044, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A044, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A1E4, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A1E4, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A1E4, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A1E4, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A010, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A010, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A34C, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A34C, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A048, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A048, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A044, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A1E4, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A1E4, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A010, 32), 32)
[WARNING ]: dynamic dst ExprMem(ExprInt(0x1006A048, 32), 32)
mapping: 0x1003f410 -> 0x100982cc with val @32[ESP_init + 0x4]: 0x21a4
skipping 0x10043850 with val None: 0xbadf00d
skipping 0x10050148 (library func)
skipping 0x10043570 with val None: 0xbadf00d
skipping 0x100505d9 (library func)
skipping 0x1005107b with val None: 0xbadf00d
skipping 0x10043500 with val None: 0xbadf00d
skipping 0x10043f80 with val None: 0xbadf00d
mapping: 0x1000f0c0 -> 0x10098589 with val @32[ESP_init + 0x8]: 0x2012
```

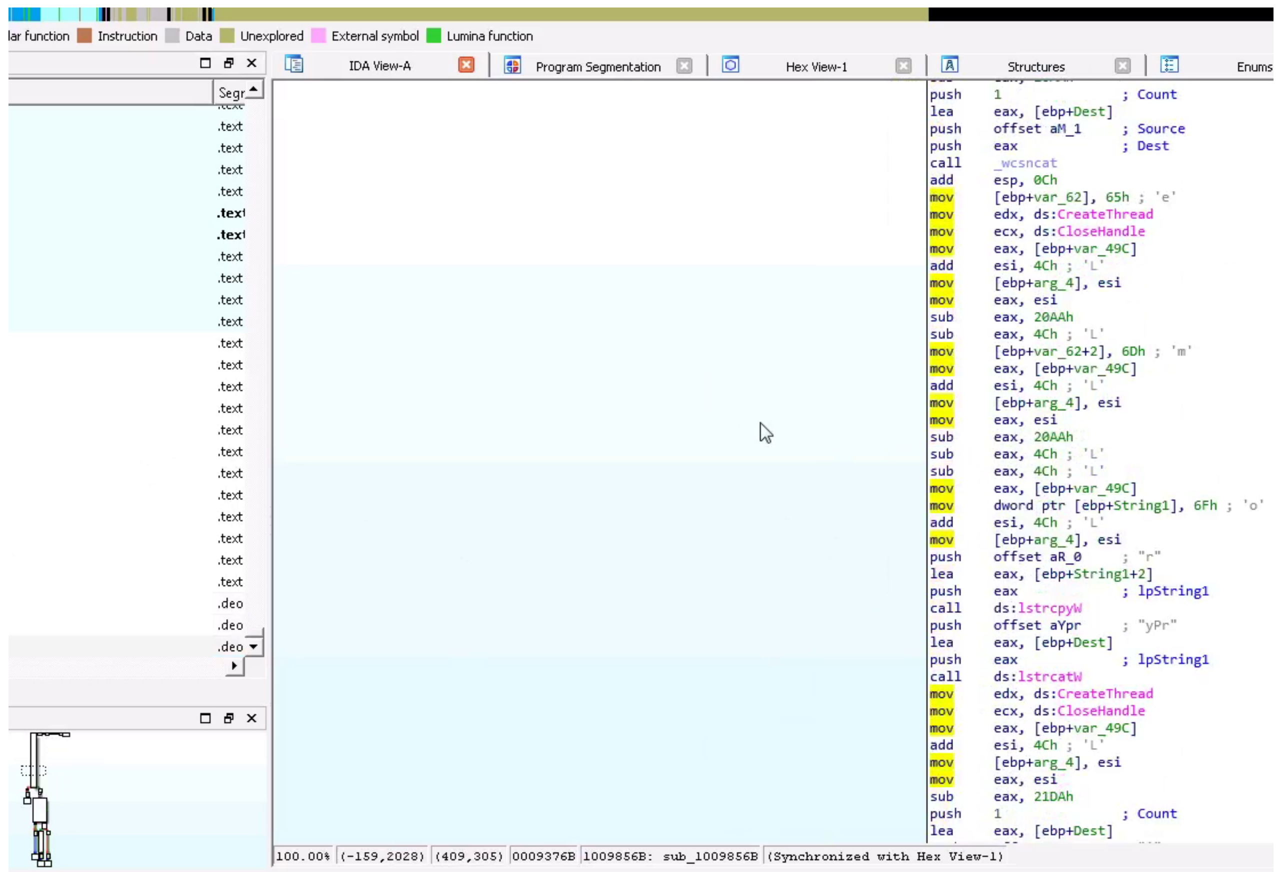
Obr. 31: Ukážka deobfuskovania identifikovaných dosiahnuteľných a obfuskovaných funkcií.

Ďalej si ukážeme náhľad jednej zo spracovaných a deobfuskovaných funkcií.

Obr. 32: Náhľad identifikovanej obfuskovanej funkcie 2.

Obr. 33: Náhľad deobfuskovanej funkcie 2.

Ďalej si ukážeme náhľad ďalšej deobfuskovanej funkcie, z ktorého je jasné, že obsahuje obfuskovaný reťazec, ktorý sa poskladá za behu.



Obr. 34: Náhľad funkcie s obfuskovaným reťazcom.

V poslednej ukážke posunieme adresy deobfuskovaných funkcií do našej metódy, ktorá sa z týchto funkcií pokúsi vytiahnuť všetky takéto reťazce.

```
user@ubuntu:~$ ipython3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: funcs = [0x10098000, 0x100982b7, 0x1009856b]

In [2]: from pprint import pformat

In [3]: from stadeo.string.string_revealer import StringRevealer

In [4]: ip = "10.1.40.164"

In [5]: sr = StringRevealer(32)

In [6]: strings = sr.process_funcs(funcs, ip=ip)
```

Obr. 35: Kód, ktorý vyhľadá obfuskované reťazce.

Jeho výsledkom je množina reťazcov poskladaných za behu v každej z poskytnutých funkcií. Taktiež je potrebné poznamenať, že počas behu môže Miasm vygenerovať viaceré varovania, ktoré nás však nezaujímajú.

```
WARNING: address 0x0 is not mapped in virtual memory:
[WARNING ]: [Errno cannot get mem ad] 0x0
[WARNING ]: cannot disasm at 0
WARNING: address 0x0 is not mapped in virtual memory:
[WARNING ]: [Errno cannot get mem ad] 0x0
[WARNING ]: cannot disasm at 0

In [7]: pformat(strings)
Out[7]: "{269058048: {'NameSourse'},\n 269058743: set(),\n 269059435: {'SeLockMemoryPrivilege'}}"

In [8]:
```

Obr. 36: Vyhľadané obfuskované reťazce.

6 Záver

V práci sme na úvod poskytli rýchly náhľad do problematiky týkajúcej sa reverzného inžinierstva a významu analýzy kódu malvéru.

Pokračovali sme vysvetlením podstaty obfuskačných techník a dôvodov, prečo bývajú aplikované na legitímny, ako aj škodlivý kód. Taktiež sme pokryli podstatu metód s presne opačným významom, a to takzvaných deobfuskačných techník.

Posledná časť úvodu zapojila do celého kontextu nakoniec rodinu malvéru známu ako Stantinko, ktorej vlastným obfuskačným technikám sme ďalej venovali pozornosť.

Nasledovala východisková kapitola, ktorá sa zaoberala teoretickými konceptami, metódami a knižnicami používanými neskôr pri deobfuskácii.

Teoretické východiská sa zaoberali najmä pojmami z teórie grafov a technikami analýzy kódu programov, u ktorých sme taktiež definovali, čo presne sú. Okrem toho táto časť pokryla aj niekoľko používaných pojmov z iných oblastí.

Technické východiská sa venovali čisto knižniciam, ktoré boli použité pri riešení. Konkrétne knižniciam Miasm – ktorá nám poskytla najmä techniky pre analýzu

kódu programov, IDAPython – ktorá doplnila nedostatky Miasm-u a umožnila nám uvažovať nad programom ako celkom a RPyC – ktorá zabezpečila jednoduchý spôsob, ako použiť predchádzajúce knižnice spoločne.

Pokračovali sme preskúmaním obfuskačných techník používaných konkrétne skupinou Stantinko. Po analýze týchto techník sme ich porovnali s podobnými a už známymi technikami, pričom sme zistili, že je medzi nimi priveľký rozdiel na to, aby sme mohli použiť už existujúce riešenia na náš problém.

Usúdili sme, ktoré techniky spôsobujú najväčšie ťažkosti a následne navrhli a implementovali riešenie, ktoré značne uľahčuje a zrýchľuje analýzu takto obfuskovaných programov.

Jednalo sa konkrétne o techniky, ktoré by sa dali považovať za variácie vyrovnaného toku riadenia a techniky s názvom stack strings.

Pri takomto vyrovnanom toku riadenia sa ukázala byť oveľa komplikovanejšia identifikácia jeho súčastí ako samotná deobfuskácia, pri ktorej sme využili väčšinu popísaných techník analýzy kódu programov a vlastností z teórie grafov.

Počas tvorby riešenia sme narazili na viaceré zákutia zvolených techník analýzy kódu programov v implementácii knižnici Miasm, ktoré sa nám podarilo prekonať. Na záver sme celý proces zrýchlili umiestnením cache na problematické miesta, čím sa náš algoritmus stal reálne použiteľným aj na najväčšie obfuskované funkcie.

Literatúra

[1] Sarfraz Khurshid, Corina S. Pasareanu, Willem Visser, Generalized Symbolic Execution for Model Checking and Testing, Cambridge, 2003

[2] James C. King, Symbolic Execution and Program Testing, IBM Thomas J. Watson Research Center, July 1976

[3] Martin Červeň, Deobfuskácia kódu generovaného programom Obfuscator-LLVM, FMFI Bratislava, 2016

[4] Jay Smith, Automatic Recovery of Constructed Strings in Malware, citované dňa:19.2.2021, dostupné online: fireeye.com/blog/threat-research/2014/08/flare-ida-pro-script-series-automatic-recovery-of-constructed-strings-in-malware.html, FireEye blog, 2014

[5] Playing with Dynamic symbolic execution, citované dňa:15.1.2021, dostupné online:miasm.re/blog/2017/10/05/playing_with_dynamic_symbolic_execution.html, Miasm blog, 2017
[6] Miasm, citované dňa:15.1.2021, dostupné online:github.com/cea-sec/miasm/tree/80e40a3d2ca735db955807ad0605b43ca22e4e35
[7] STANISLAV MELO, ZRYCHLENÍ VYKONÁVÁNÍ SOFTWARU POMOCÍ AUTOMATICKÝCH INSTRUKČNÍCH ROZŠÍŘENÍ, Fakulta informačních technologií, VUT Brno, 2013
[8] Vladislav Hrčka, Stantinko's new cryptominer features unique obfuscation techniques, citované dňa:31.1.2021, dostupné online:welivesecurity.com/2020/03/19/stantinko-new-cryptominer-unique-obfuscation-techniques/, ESET blog, 2020
[9] Ken Kennedy, A survey of data flow analysis techniques, IBM Thomas J. Watson Research Division, 1979
[10] Preston Briggs, Tim Harvey, Computing dominators and dominance frontiers, Technical report, Rice University, 1994
[11] Vaibhav Jaimini, Articulation Points and Bridges, citované dňa:15.1.2021, dostupné online:hackerearth.com/practice/algorithms/graphs/articulation-points-and-bridges/tutorial/
[12] Nelson, REMOTE PROCEDURE CALL, Carnegie Mellon University, 1982
[13] Hasabnis N, Sekar R, Lifting assembly to intermediate representation: A novel approach leveraging compilers, InProceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, Strany 311-324, 2016
[14] Robin Eklind, LLVM IR and GO, citované dňa:15.1.2021, dostupné online:blog.gopheracademy.com/advent-2018/llvm-ir-and-go, 2018
[15] RPyC documentation, dostupné online:rpyc.readthedocs.io/en/latest/docs.html
[16] IDAPython documentation, citované dňa:15.1.2021, dostupné online:hex-rays.com/products/ida/support/idapython_docs/
[17] Data flow analysis: DepGraph, citované dňa:15.1.2021, dostupné online:miasm.re/blog/2017/02/03/data_flow_analysis_depgraph.html#showcase-2-tracking-arguments-on-the-stack, Miasm blog, 2017
[18] Reaching Definitions, citované dňa:15.1.2021, dostupné online:github.com/cea-sec/miasm/blob/eae166b6d703e9c394e1fd00e98328289192a12d/miasm/analysis/data_flow.py#L23
[19] DiGraphDefUse, citované dňa:15.1.2021, dostupné online:github.com/cea-sec/miasm/blob/eae166b6d703e9c394e1fd00e98328289192a12d/miasm/analysis/data_flow.py#L120
[20] Lowering to LLVM and CodeGeneration, citované dňa:15.1.2021, dostupné online:mlir.llvm.org/docs/Tutorials/Toy/Ch-6/

[21] Niranjan Hasabnis, R. Sekar, Intel, Stony Brook University, Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers, ASPLOS '16, Atlanta, GA, USA, 2016
[22] Chakravarty MM, Keller G, Zadarnowski P, A functional perspective on SSA optimisation algorithms. Electronic Notes in Theoretical Computer Science, 2004
[23] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, Rahul Purandare, Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks, Under Review IEEE Transactions on Software Engineering, 2020.