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

...PODAKOVANIE

...abstrakt

...abstraCt

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....8
1.2 Technologické východiská....9
1.2.1 Miasm....9
1.2.2 IDAPython....15
1.2.3 RPyC....16
2 Obfuskačné techniky skupiny Stantinko – analýza....17
3 Popis podobných známych obfuskačných techník....17
4 Deobfuskácia – popis implementácie....17
5 Záver....17
Literatúra....17

Obrázky

Obr. 1: Ukážka disassemblovania kódu pomocou Miasm-u.....11
Obr. 2: Ukážka assemblovania kódu pomocou Miasm-u.....12
Obr. 3: Ukážka symbolického vykonávania pomocou Miasm-u.....13
Obr. 4: Ukážka emulácie kódu pomocou Miasm-u.....14
Obr. 5: Ukážka pripájania dynamického symbolického vykonávania k Sandbox-u v Miasm-e.....14
Obr. 6: Ukážka inicializácie RPyC servera.....17
Obr. 7: Ukážka pripojenia k RPyC serveru.....17

Úvod

Analýza malvéru je proces rekonštrukcie podozrivých súborov a porozumeniu im, zvyčajne za účelom vyhodnotenia dopadu kybernetických útokov alebo nájdenia a extrahovania unikátnych prvkov a vlastností. Pomocou získaných údajov je možné identifikovať podobné súbory, ktoré boli videné v minulosti, alebo naopak rozoznať ich, keď sa ukážu v budúcnosti. Na základe týchto indikátorov sa dá nakoniec napríklad odhaliť iné, už prebiehajúce, 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 a tým 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 poslednej štvrtej kapitole opíšeme samotnú implementáciu výsledného kódu.

1 Východiská

1.1 Teoretické východiská

V tejto kapitole si priblížime teoretické východiská bakalárskej práce.

1.1.1 Teória grafov

Strom. Strom je v teórii grafov taký graf, v ktorom akékoľvek dva vrcholy sú 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

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*, ktorý 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, 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 z nich naopak vystupujú. [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, program counter 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, pokúsi sa 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 ich konkrétnymi hodnotami alebo získaním konkrétnych hodnôt po systémovom alebo knižničnom volaní, na druhú stranu môže ale stratiť niektoré možné no komplikované cesty. [5]

1.1.3 Iné teoretické východiská

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 následujúca inštrukcia v pamäti.

Základný blok. Pod základný blokom chápeme 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.

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 nepokrytý základný blok, skok alebo cestu [5]

Napriek tomu má Miasm ale stále niekoľko nedostatkov, ako napríklad to, že nedokáže spracovať jump tabuľky a uvažovať nad celým programom ako celkom, a teda trebárs rozlíšiť funkcie bez návratu alebo sám zistiť výšku zásobníku po volaní ľubovolnej funkcie, preto 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 13 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 13. 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 callback *dis_block_callback*, ktorý je volaný po každom disassemblovanom bloku, jeho parametre sú samo vysvetľujúce.

```
1. from miasm.analysis.binary import Container
2. from miasm.analysis.machine import Machine
3. from miasm.core.locationdb import LocationDB
4.
5. def dis_callback(mdis, cur_block, offsets_to_dis):
6.         return
7.
8. cont = Container.from_stream(open('file.bin', 'rb'))
9. machine = Machine(cont.arch)
10. unreachable = []
11. mdis = machine.dis_engine(cont.bin_stream, loc_db=LocationDB(), dont_dis=unreachable)
12. mdis.dis_block_callback = dis_callback
13. 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;* assemblovanie ale 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 offset-y 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 19 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 a 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. new_next_addr_card = ExprLoc(asmcfg.loc_db.get_or_create_name_location('_'), 64)
8. for block in asmcfg.blocks:
9.         for instr in block.lines:
10.            for ind in range(len(instr.args)):
11.                if rip in instr.args[ind]:
12.                    next_addr = ExprInt(instr.offset + instr.l, 64)
13.                    fix_dict = {rip: rip + next_addr - new_next_addr_card}
14.                    instr.args[ind] = instr.args[ind].replace_expr(fix_dict)
15.        if asmcfg.loc_db.get_location_offset(block.loc_key):
16.                asmcfg.loc_db.unset_location_offset(block.loc_key)
17.
18. asmcfg.loc_db.set_location_offset(asmcfg.heads()[0], out_addr)
19. patches = asm_resolve_final(machine.mn, asmcfg,
20.                             asmcfg.loc_db, 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 point-om.

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 pričom registre, ktoré majú byť clobbered podľa ABI a stack pointer, označí 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 pokiaľ je adresa nasledujúceho bloku, podľa aktuálneho symbolického stavu, jednoznačná a jej prislúchajúci blok preložený do Miasm IR.

```
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, ktorá 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*, 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, ich názvy 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 a tým 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 jej je možné posunúť niekoľko voliteľných parametrov:

- *implicit* – flag, ktorý keď je nastavený, symbolický program counter je pridaný medzi sledované výrazy pri každom prechádzanom bloku kódu
- *apply_simp* – flag, ktorý keď je nastavený, na spracovávané výrazy sú aplikované simplifikačné pravidlá, teda napríklad v prípade, že sleduje register *EAX* a narazí na inštrukciu *XOR EAX, EAX*, bude vedieť, že do registra *EAX* bude vždy priradená nula a nie je potrebné ďalej sledovať *EAX*
- *follow_mem* – flag, ktorý keď je nastavený, operácie s pamäťou a všetky výrazy, od ktorých závisia sú pridané medzi sledované
- *follow_call* – flag, ktorý keď je nastavený, všetky volania a výrazy, od ktorých závisia sú pridané medzi sledované

Metóda *get()* následne generuje výsledky pre dané elementy konkrétneho riadku, posledný parameter je pritom 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 dané hodnoty pre hľadané elementy podľa aktuálneho grafu, ktoré avšak môžu byť aj symbolické. V prípade, že bol nastavený parameter *implicit*, dokážeme metódou *is_satisfiable()* zistiť či existuje nejaký model pre dané symbolické obmedzenia, a ak nejaký existuje, metódou *consraints()* ľubovoľný model 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 aspoň za predpokladu, že sama bola schopná spracovať program bez problémov, na druhú stranu jej chýba 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 stack pointer-u 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 *ea2* je rovné ‘BADADDR‘, pokúsi sa rozlíšiť koniec funkcie samostatne
- *idaapi.FUNC_LIB* – flag určujúci či je daná funkcia knižničná

[16]

1.2.3 RPyC

RPyC, alebo Remote Python Call, je knižnica pre programovací jazyk Python predovšetkým na symetrické vzdialené volania procedúr, ktoré umožňujú pracovať s vzdialenými objektmi, 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 a zopakujeme príkaz s výrazom ‘hi here‘ na riadku 7, tento krát sa výraz zobrazí u klienta, následne 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 Obfuskačné techniky skupiny Stantinko – analýza

3 Popis podobných známych obfuskačných techník

4 Deobfuskácia – popis implementácie

5 Záver

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

[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

[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