Эвристический метод поиска уязвимостей в ПО без использования исходного кода

А.С. Велижанин, А.В. Ревнивых

Тюменский государственный нефтегазовый университет
e-mail: Anatoliy.Velizhanin@gmail.com
Тюменский государственный нефтегазовый университет, ИВТ СО РАН
Alexchr@mail.ru

Аннотация

В работе рассмотрены подходы поиска уязвимостей в программном обеспечении современных ИВС. Предложен эвристический метод автоматизированного поиска уязвимостей в ПО без использования исходного кода.

Современное программное обеспечение создается на основе сложной комбинации множества различных технологий и работает в гетерогенной среде, состоящей из всевозможных видов устройств. Объем отдельных программных модулей в составе системы зачастую превышает миллионы строк кода на высокоуровневых языках программирования. Так, например, одно только ядро ОС Linux 3.5.4 в архиве занимает около 77.2 Мб и более 500Мб в распакованном виде. Однако ядро любой ОС является не единственным ее компонентом. Современные дистрибутивы Linux сочетают в себе целый набор различного ПО. Сюда входит, например, базовый набор программ, работающих в терминале и обеспечивающих минимально необходимое взаимодействие с ОС (одним из простейших примеров подобного рода программного модуля, используемого для построения встраиваемых систем, можно считать BusyBox [1]). Кроме того, современные ОС предоставляют пользователю графическое окружение. Для UNIX-подобных операционных систем таких графических оболочек имеется множество. Некоторыми примерами могут быть GNOME, KDE, XFCE и др. Однако данные графические оболочки так же не функционируют самостоятельно на базе ядра ОС, а требуют для своей работы X Window System (например, X.Org и т.п.). Помимо стандартного набора программ, пользователь, как правило, устанавливает различное дополнительное ПО. Аналогичная ситуация сложности и гетерогенности системы наблюдается и в других ОС, таких как Microsoft Windows, а так же во все нарастающей популярности платформ для мобильных устройств (например, Apple iOS и Google Android). Рост популярности какого-либо программного продукта ведет к его активному усложнению, а при возможности и портированию под другие платформы, что еще больше усложняет данный программный продукт. Рассматривая прикладное ПО мы видим, что его функционирование основано на множестве других программных модулей, в том числе модулей ОС. Так, для обычного открытия файла прикладное ПО вызывает соответствующую API функцию ОС, которая в свою очередь

генерируют системный вызов, инициирует переключение контекста и, в конечном счете, приводит к активности системных драйверов, ранее маппированных в процесс System (для ОС Microsoft Windows). Мы можем увидеть подобную активность, набрав в командной строке «dir C:\ /s», что приведет к активной работе с диском «C:», а в «Process Explorer», входящем в поставку утилит Sysinternals [2], выбрав процесс System, открыть его свойства (пункт Properties) и отсортировав по полю «CSvitch Delta» (переключения контекста). Далее мы увидим в списке используемые стартовые адреса вызовов. Выделив нужный пункт, мы можем посмотреть какому именно модулю принадлежит данный вызов, нажав на кнопку «Module». Кроме того, в ОС Microsoft Windows драйвера так же могут быть как режима ядра, так и пользовательского режима (KMDF [3] и UMDF [4], соответственно), а страницы в памяти, на которых расположены эти драйвера, как выгружаемыми в файл подкачки, так и нет. Драйвера UMDF используют ALPC для взаимодействия с модулями ядра для реального физического доступа к устройству. А запуск любой программы в Windows приводит к инициализации соответствующей подсистемы для запускаемого процесса, настройки окружения и множеству других системных событий [5]. Кроме того, производственными объектами так же управляют компьютерные системы, которые, как правило, используют ОС реального времени для автоматизации технологических процессов. Ситуация усложняется еще и тем, что современные компьютерные системы активно используют различные технологии сетевых коммуникаций. Активное развитие облачных технологий приводит к еще большей интеграции отдельных компьютерных систем друг в друга.

Помимо общей программно-аппаратной загруженности современных компьютерных систем в современном мире приходится сталкиваться с необходимостью постоянного обеспечения ИБ как целых компьютерных инфраструктур, так и обычных персональных компьютеров, планшетов, ноутбуков, смартфонов и т.п.

Для обеспечения ИБ используются следующие механизмы:

1. Управление доступом (избирательное, мандатное или на основе ролей)
2. Защита самой информации (шифрование, стеганография и т.п.)

Однако, несмотря на существующие меры, направленные на обеспечение защищенности информационно-вычислительных систем и сетей, существует множество инцидентов нарушения ИБ. Наиболее яркими из последних случаев мы можем отметить деятельность группировок Anonimous [9], LulzSec[10], а так же появление вредоносного ПО для компьютерных систем, такое как Stuxnet [11], Flame [12], Mac.Iservice впоследствии сформировавший iBot [13], заражение корневого репозитория Linux www.kernel.org [14]. Так же периодически появляются сообщения об уязвимостях Google Android, а так же появлении вредоносного ПО в Google Android Market и Apple Mac App Store. Помимо прочего регулярно появляется информация об обнаруженных уязвимостях в самом различном программном обеспечении.

Таким образом, мы имеем нашумевшие случаи нарушения ИБ фактически во всех наиболее распространенных ОС и прикладных программных продуктах. Следует отметить, что отсутствие столь большого количества информации по случаям компрометации компьютерных систем с отличным системным и прикладным программным обеспечением не означает абсолютной их защищенности. Конечно, уровень вероятности успешной атаки обычным злоумышленником на ОС, применение которой чрезвычайно редко и о которой, возможно, злоумышленник ни разу не слышал, низок. Однако следует учитывать тенденции последних лет, которые подарили миру компьютерных технологий вредоносное ПО наподобие Flame, DuQu, Gauss и т.п., чья активность показывает, что их создание, вероятно, велось целой группой высокопрофессиональных разработчиков в сфере безопасности компьютерных систем и методам защиты соответствующей ОС, вероятно, в государственных масштабах [15].

Следует учесть, что подобного рода инциденты возникли по причине наличия уязвимостей в соответствующих компьютерных системах. Рассматривая вопрос уязвимости ИВС, мы можем выделить следующие составляющие:

1) Архитектурные уязвимости
2) Уязвимости реализации

Далее в работе мы будем рассматривать именно уязвимости в ПО, связанные с его реализацией.

Рассматривая программы, написанные на языке программирования C/C++, мы можем отметить следующие наиболее известные типы уязвимостей:

1) Уязвимости функций копирования массивов данных (как стандартной библиотеки языка, так и собственных реализаций смежных задач)
2) Уязвимости форматных строк
3) Гонки

Вкратце рассмотрим каждый из приведенных типов уязвимостей.

Уязвимости функций копирования массивов данных связаны с тем, что язык программирования C/C++ не производит контроля длины массивов при выполнении каких-либо операций с ним. Приведем простой поясняющий пример в листинге 1.

Листинг 1.

```
#include <iostream>
#include <string>

#define MAX_PARAMLENGTH 10

using namespace std;

int main(int argc, char *argv[])
{
```

```
        if (argc < 2)
        {
                cout << "Start by: <programm_name.exe> <string_parameter>\n";
                return -1;
        }

        char arr0[MAX_PARAMLENGTH + 1];
        char arr[MAX_PARAMLENGTH + 1];
        char arr1[MAX_PARAMLENGTH + 1];
        strcpy(arr, argv[1]);

        cout << arr << endl;

        return 0;
}
```

При выполнении данной программы, если в качестве аргумента мы укажем слишком длинную строку, то мы получим аварийное завершение работы программы. Однако, помимо прочего, в общем случае мы можем так же выполнить произвольный код с правами пользователя, от имени которого работает данная программа, если, например, запишем адрес, соответствующий нашему shell-коду в место в стеке, хранящее адрес возврата из функции и т.п. Конечно же, существует ряд технологий противодействия подобного рода эксплуатации уязвимостей (ASLD, DEP и т.п.), однако, в ряде случаев, имеются пути обхода подобного рода защит (например, если приложение использует не поддерживающую ASLR библиотеку, то мы можем передать управление на нужную нам последовательность инструкций в ней; некоторые приложения не могут запретить выполнение кода в куче по причине того, что в часть их функционала входит динамическая кодогенерация или выполнение скриптов; так же имеются и другие технологии обхода подобных защит). Таким образом, несмотря на то, что эксплуатация подобных уязвимостей несколько усложнилась, она не была ликвидирована полностью.

Уязвимости форматных строк предоставляют возможность чтения или записи значений из определенных адресов внутри адресного пространства, доступного уязвимому процессу. Несколько модифицируем пример из листинга 1 в листинге 2.

Листинг 2.

```
#include <iostream>
#include <string>

#define MAX_PARAMLENGTH 100

using namespace std;

int main(int argc, char *argv[])
{
        if (argc < 2)
        {
                cout << "Start by: <programm_name.exe> <string_parameter>\n";
                return -1;
        }

        char arr0[MAX_PARAMLENGTH + 1];
        char arr[MAX_PARAMLENGTH + 1];
```

```
        char arr1[MAX_PARAMLENGTH + 1];
        strcpy(arr, argv[1]);

        printf(arr);

        return 0;
}
```

В данном примере, оперируя модификаторами «%X» и «%n» злоумышленник может либо вывести некоторую область адресного пространства процесса, либо произвести запись. Данное действие может привести не только сбою в работе программы, но и стать причиной утечки важной информации (например, если злоумышленник таким способом сможет получить пароль на доступ, например, к базе данных, если он хранится в оперативной памяти данной программы в незашифрованном виде) или незаметной модификации злоумышленником необходимых значений внутри доступного адресного пространства уязвимого процесса.

Под гонками мы понимаем ситуацию, когда некоторый процесс, например, работающий с закрытой от пользователя информацией, создает множество временных файлов. Однако операции создания и открытия файла не атомарны (за исключением случая использования системных вызовов или API, которые создают файл и сразу его открывают). Кроме того, современные ОС поддерживают функционирование множества процессов параллельно. Таким образом, в некоторых случаях мы имеем потенциальную возможность успеть записать некоторые данные в файл, созданный уязвимым к данному типу атаки ПО.

Отметим, что, несмотря на обилие различных языков программирования, в конечном счете, все они связаны с соответствующим RTL и API ОС. Языки программирования, использующие JIT компиляцию, так же создают ПО, в конечном счете, выполняющее вызовы API ОС и преобразованное в машинный код JIT компилятором. Таким образом, данные языки программирования несколько абстрагируют использующего их разработчика от деталей работы компьютерной системы и ОС, однако не решают проблему, отложив ее на уровень виртуальной машины, JIT компилятора, библиотек базовой библиотеки классов языка программирования и т.п.

Все вышеперечисленное подтверждает необходимость разработки средств поиска уязвимостей. В большинстве своем данные средства опираются на анализ исходного кода и выдачу ряда предупреждений компилятором. Так, например, имеется ряд статических анализаторов типа PVS_Studio [16]. Конечно, имеются средства наподобие Application Verifier, однако по-прежнему среди средств и методов поиска уязвимостей в ПО без использования исходного кода имеется только ручной поиск с использованием дизассемблеров, некоторые продукты для сигнатурного анализа бинарных файлов и метод Fuzzing-a.

В условиях гетерогенности современных компьютерных систем и сложности их интеграции друг в друга ручной поиск уязвимостей является крайне сложным и поэтому относительно малоэффективным. Однако именно человек способен провести качественный и по возможности многофакторный анализ сложных многоступенчатых алгоритмов возникновения

возможных уязвимостей. Сигнатурный анализ бинарных файлов так же осложнен рядом причин. Например, использование различных компиляторов и комбинаций параметров сборки приложения может привести к абсолютно различному коду. Имеется технология FLIRT [17], позволяющая провести анализ ПО и поиск стандартных функций по их сигнатурам, которая согласно источнику [17] дает вполне качественные результаты, однако ее применение специализировано. Fuzzing - является одним из методов анализа ПО, ориентированный именно на работу с бинарными файлами. Принцип его действия основан на передаче произвольных (или почти произвольных) аргументов на точки входа в процесс в ожидании возникновения исключительной ситуации или некорректного поведения ПО. Следует отметить, что автоматически определить корректно ли работает ПО довольно сложно, поэтому, в общем случае, ожидается возникновение исключительной ситуации в ходе анализа.

Таким образом, мы фактически не имеем средства автоматизированного анализа ПО без использования исходных кодов, которое бы было хотя бы частично лишено недостатков вышеописанных средств. Традиционно из методов поиска уязвимостей в ПО выделяют следующие категории:

1) Метод белого ящика
2) Метод черного ящика
3) Метод серого ящика

Метод белого ящика подразумевает под собой поиск уязвимостей, опираясь на исходный высокоуровневый код ПО.

Метод черного ящика является способом анализа поведения программы, не исследуя ее внутренней структуры. Простой Fuzzing является подходящим примером такого анализа. Уточним, что под простым Fuzzing-ом мы понимаем непосредственный тест без предварительного изучения объекта, что не всегда является реалиями процесса Fuzzing-а. Так, исследователь зачастую предварительно анализирует программный модуль в поисках наиболее подходящего направления теста. Однако, даже подобный анализ, если он не дает представлений о внутренней структуру и алгоритмах функционирования ПО, в принципе, наверное, можно назвать тестом методом черного ящика.

Метод серого ящика – является промежуточным способом между методами черного и белого ящика. Чаще всего данный метод применяется при анализе ПО экспертом. Как правило, для осуществления качественного теста данным методом необходимы знания языка программирования Assembler для соответствующей архитектуры процессора и правил функционирования ПО в данной среде. Под данной средой мы понимаем непосредственное окружение ПО. В большинстве случаев таким окружением будет являться ОС, а точнее ее подсистемы Executive и Environment. Мы указываем именно Executive, поскольку именно данная подсистема ОС имеет связь со всеми другими подсистемами и является своеобразной прослойкой между Kernel Mode и подсистемами из User Mode. Подсистема Environment указана потому, что данная подсистема предоставляет окружение запускаемому процессу. В Microsoft

Windows 7 мы имеем подсистему Win32, подсистему OS/2 и подсистему POSIX [5]. Примеры подсистем приводились для ОС Microsoft Windows. Альтернативные ОС могут иметь несколько отличную архитектуру. Однако, несмотря на сложность данного метода, он сочетает в себе положительные стороны обоих методов. Конечно, дизассемблированное представление ПО является далеким от высокоуровневой исходного кода и значительная часть информации о реализации ПО теряется. Так, например, дизассемблировав Native программный модуль мы уже не сможем его обратно собрать с помощью обычных транслятора и линкера. Отметим, что байт-код, например языка программирования Microsoft C#, лишен данного недостатка и позволяет декомпилировать исполняемый модуль прямо в высокоуровневый код на языке программирования Microsoft C#. Однако, возвращаясь к рассмотрению именно Native кода, отметим, что, несмотря на невозможность сборки дизассемблированного кода в корректно работающий исполняемый модуль без предварительной серьезной обработки и потерю значительного количества информации о структуре программы (например, мы не можем однозначно определить границы двух близлежащих массивов символов в стеке), мы можем восстановить некоторые алгоритмы работы данного программного модуля.

Генерируемый дизассемблерами код различен по внешней структуре. Потому важным является формирование требуемой структуры дизассемблированного листинга для упрощения парсинга эвристическим анализатором. Приведем один из возможных вариантов структуры дизассемблированного листинга:

1) Адрес в формате «0123456789ABCDEF» (пример для 64-битных систем; возможно наличие префикса 0x)
2) Символ «:» (генерируется многими дизассемблерами; удобен как разделитель при парсинге)
3) Шестнадцатеричный код инструкции в формате «E9 12 02 00 00» (обычно каждый код дизассемблерами разделяется символом пробела)
4) Инструкция процессора в виде мнемонического кода типа «mov»
5) Список аргументов через запятую (учитывая, что количество аргументов может быть более 2, например, в команде imul и т.п.)
6) Возможное наличие символа «;» и комментария в продолжении строки

Данная структура является стандартной для языка программирования Assembler. Парсинг такого кода является вполне простой задачей.

Рассматривая работу дизассемблеров, мы видим, что они оперируют высокоуровневым понятием «функция» при предоставлении дизассемблированного кода в виде блоков (мы можем увидеть это на примере дизассемблера Hex Rays Ida Pro). Данный подход удобен при анализе человеком, однако формирует крупные блоки кода со сложными внутренними зависимостями. Внутренние зависимости крупных блоков могут в значительной степени повлиять на результат работы даже отдельно взятой функции. Однако язык программирования Assembler в своем стандарте изначально не содержал понятия функции, которое было добавлено в качестве

расширений различными реализациями трансляторов. Для автоматизированного анализа становится наиболее подходящим мелкое дробление дизассемблированного кода в такой степени, чтобы построить максимально детальную структуру исследуемого кода. В таком случае мы можем разбить дизассемблированный код на блоки с линейным ходом выполнения, только последняя инструкция которых, может быть инструкцией перехода. Тогда мы получим максимально детальную карту переходов исследуемого программного модуля.

Рассмотрим простейший пример. Для повышения наглядности используем новый листинг с чуть большим числом переходов на пути к потенциально уязвимому блоку кода. Пример приведен в листинге 3.

Листинг 3.

```
#include <iostream>
#include <string>

#define MAX_PARAMLENGTH 10

using namespace std;

void evil_func(char *evilC)
{
        char arr0[MAX_PARAMLENGTH + 1];
        char arr[MAX_PARAMLENGTH + 1];
        char arr1[MAX_PARAMLENGTH + 1];

        strcpy(arr, evilC);
}

int main(int argc, char *argv[])
{
        if (argc < 2)
        {
                cout << "Start by: <programm_name.exe> <string_parameter>\n";
                return -1;
        }

        evil_func(argv[1]);

        cout << "Evil_func has done his business!" << endl;

        return 0;
}
```

Карту переходов можно представить в виде графа, а так же в виде матрицы. В листинге 4 представлен отрывок дизассемблированного кода. В таблице 1 представлен результат автоматизированного средства разбиения кода на блоки.

Листинг 4.

```
00000001400010FF: E9 0C 03 00 00     jmp         0000000140001410
0000000140001410: 48 89 4C 24 08     mov         qword ptr [rsp+8],rcx
0000000140001415: 57                 push        rdi
0000000140001416: 48 81 EC C0 00 00  sub         rsp,0C0h
000000014000141D: 48 8B FC           mov         rdi,rsp
0000000140001420: B9 30 00 00 00     mov         ecx,30h
```

```
000000014000142A: F3 AB              rep stos    dword ptr [rdi]
000000014000142C: 48 8B 8C 24 D0 00  mov         rcx,qword ptr [rsp+0D0h]
0000000140001434: 48 8B 05 E5 EC 00  mov         rax,qword ptr [__security_cookie]
000000014000143B: 48 33 C4           xor         rax,rsp
000000014000143E: 48 89 84 24 B0 00  mov         qword ptr [rsp+0B0h],rax
0000000140001446: 48 8B 94 24 D0 00  mov         rdx,qword ptr [rsp+0D0h]
000000014000144E: 48 8D 4C 24 58     lea         rcx,[rsp+58h]
0000000140001453: E8 42 30 00 00     call        strcpy
000000014000145B: 48 8D 15 3E B8 00  lea         rdx,[14000CCA0h]
0000000140001462: E8 99 30 00 00     call        _RTC_CheckStackVars
0000000140001467: 48 8B 8C 24 B0 00  mov         rcx,qword ptr [rsp+0B0h]
0000000140001472: E8 E9 33 00 00     call        __security_check_cookie
0000000140001477: 48 81 C4 C0 00 00  add         rsp,0C0h
000000014000147E: 5F                 pop         rdi
000000014000147F: C3                 ret
0000000140001490: 48 89 54 24 10     mov         qword ptr [rsp+10h],rdx
0000000140001495: 89 4C 24 08        mov         dword ptr [rsp+8],ecx
0000000140001499: 57                 push        rdi
000000014000149A: 48 83 EC 20        sub         rsp,20h
000000014000149E: 48 8B FC           mov         rdi,rsp
00000001400014A1: B9 08 00 00 00     mov         ecx,8
00000001400014AB: F3 AB              rep stos    dword ptr [rdi]
00000001400014AD: 8B 4C 24 30        mov         ecx,dword ptr [rsp+30h]
00000001400014B1: 83 7C 24 30 02     cmp         dword ptr [rsp+30h],2
00000001400014B6: 7D 1A              jge         00000001400014D2
00000001400014B8: 48 8D 15 B1 B6 00  lea         rdx,[14000CB70h]
00000001400014BF:    48    8B    0D    F2    11    01       mov                                 rcx,qword    ptr
[__imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A]
00000001400014C6:       E8       48       FC       FF       FF                                                call
@ILT+270(??$?6U?$char_traits@D@std@@@std@@YAAEAV?$basic_ostream@DU?$char_traits@D@std@@@@AEAV10@PEBD@Z)
00000001400014CB: B8 FF FF FF FF     mov         eax,0FFFFFFFFh
00000001400014D0: EB 3C              jmp         000000014000150E
00000001400014D2: B8 08 00 00 00     mov         eax,8
00000001400014D7: 48 6B C0 01        imul        rax,rax,1
00000001400014DB: 48 8B 4C 24 38     mov         rcx,qword ptr [rsp+38h]
00000001400014E0: 48 8B 0C 01        mov         rcx,qword ptr [rcx+rax]
00000001400014E4: E8 16 FC FF FF     call        00000001400010FF
00000001400014E9: 48 8D 15 B8 B6 00  lea         rdx,[14000CBA8h]
00000001400014F0:    48    8B    0D    C1    11    01       mov                                 rcx,qword    ptr
[__imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A]
00000001400014F7:       E8       17       FC       FF       FF                                                call
@ILT+270(??$?6U?$char_traits@D@std@@@std@@YAAEAV?$basic_ostream@DU?$char_traits@D@std@@@@AEAV10@PEBD@Z)
00000001400014FC:    48    8B    15    ED    10    01       mov                                 rdx,qword    ptr
[__imp_?endl@std@@YAAEAV?$basic_ostream@DU?$char_traits@D@std@@@1@AEAV21@@Z]
0000000140001503: 48 8B C8           mov         rcx,rax
0000000140001506:    FF    15    F4    10    01    00       call                                    qword    ptr
[__imp_??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAAAEAV01@P6AAEAV01@AEAV01@@Z@Z]
000000014000150C: 33 C0              xor         eax,eax
000000014000150E: 48 83 C4 20        add         rsp,20h
0000000140001512: 5F                 pop         rdi
0000000140001513: C3                 ret
```

Таблица 1.

```
-Matrix- #0
  -Block- #0 Start=0x400010FF End=0x400010FF
  -Block- #1 Start=0x40001410 End=0x40001453
  -Block- #2 Start=0x4000145B End=0x40001462
  -Block- #3 Start=0x40001467 End=0x40001472
  -Block- #4 Start=0x40001477 End=0x4000147F
  -Block- #5 Start=0x40001490 End=0x400014B6
  -Block- #6 Start=0x400014B8 End=0x400014C6
  -Block- #7 Start=0x400014CB End=0x400014D0
  -Block- #8 Start=0x400014D2 End=0x400014E4
  -Block- #9 Start=0x400014E9 End=0x400014F7
  -Block- #10 Start=0x400014FC End=0x40001506
  -Block- #11 Start=0x4000150C End=0x40001513
```

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
(0)	0	0	0	0	0	0	0	0	1	0	0	0

(1)	1	0	0	0	0	0	0	0	0	0	0	0
(2)	0	2	0	0	0	0	0	0	0	0	0	0
(3)	0	0	2	0	0	0	0	0	0	0	0	0
(4)	0	0	0	2	0	0	0	0	0	0	0	0
(5)	0	0	0	0	0	0	0	0	0	0	0	0
(6)	0	0	0	0	0	1	0	0	0	0	0	0
(7)	0	0	0	0	0	0	2	0	0	0	0	0
(8)	0	0	0	0	0	1	0	0	0	0	0	0
(9)	4	0	0	0	0	0	0	0	0	0	0	0
(10)	0	0	0	0	0	0	0	0	0	2	0	0
(11)	0	0	0	0	0	0	0	1	0	0	2	0

В таблице 1 синим цветом выделен потенциально опасный блок кода (блок 1). Желтым цветом – пути перехода в потенциально опасный блок. Оранжевым цветом отмечен блок входа, являющийся точкой входа для анализа (функция main языка программирования C/C++). Зеленым отмечен возможный альтернативный путь. Значение «1» в данной таблице соответствует переходу на указанный блок, значению «2» - внешний вызов call, который предположительно (согласно стандарту) вернет управление на следующую инструкцию. Значению «3» соответствует переход по внешнему адресу, возврат из которого мы не можем предположить автоматически. Значение «4» соответствует возврату из вызова, совершенного по адресу «00000001400014E4». Данная матрица является расширенной матрицей смежности, хранящей в себе помимо информации о переходах так же информацию о механизмах этих переходов. Информационное наполнение матрицы является расширяемым.

Потенциально уязвимым является блок кода №1, содержащий в себе функцию стандартной библиотеки языка программирования C/C++ «strcpy».

Поворотными точками будут являться адреса инструкций условного перехода, которые имеют непосредственное отношение для достижения заданного потенциально-опасного блока кода. На рассматриваемой матрице мы можем их определить, ориентируясь на ячейки, отмеченные желтым цветом. Столбцы, соответствующие данным ячейкам, представляют собой отдельные блоки кода с линейным ходом выполнения. Если столбец содержит значение «1» в двух ячейках (согласно инструкциям рассматриваемого набора процессоров команды условного перехода могут передавать управление только на 2 адреса: либо на адрес, указанный в качестве единственного аргумента, либо на следующий за данной инструкцией адрес), то мы имеем условную инструкцию. Из приведенной матрицы мы имеем путь: 5->8->0->>>1. Задействованный условный переход мы имеем в блоке кода №5.

Рассматриваемый пример программы и, соответственно, дизассемблированного листинга не большой. В реальных проектах размер листинга может быть значительно больше, что может значительно сказаться на времени анализа. Для уменьшения затрат времени мы можем внести коэффициент прохода в глубину. Тогда из каждой точки входа информации в процесс, мы построим возможные варианты перехода на указанную глубину, а так же построим возможные пути до потенциально уязвимых блоков кода так же указанной глубиной (для осуществления данного построения можно использовать обратный проход от потенциально опасного блока, ориентируясь на построенную матрицу смежности). Тогда пересечение множеств, полученных,

при таком анализе, даст варианты путей достижения потенциально опасного блока. В некоторых случаях можно увеличивать глубину анализа. Данный метод поиска путей является эвристическим и не дает всех возможных вариантов. Построение всевозможных вариантов путей для реальных приложений является чрезвычайно затратной по времени задачей, поскольку зависимость по времени от прироста числа блоков не линейна. Использование матрицы достижимости, которая строится по простой матрице смежности, в реальных проектах так же не представляется целесообразным, поскольку требует возведения математических матриц в высокие степени, что так же требует значительных временных затрат (особенно в тех случаях, когда матрицы имеют высокие размерности)

Альтернативой матричному подходу может являться подход из теории графов. Данный подход повлечет за собой только изменение структуры хранения информации и, соответственно, несколько отличные методы поиска и не повлечет коренных изменений сути эвристического метода.

Опираясь на полученные в ходе анализа пути, мы можем выбрать перспективные вектор дальнейшего исследования. Под вектором исследования мы понимаем определенный путь потока выполнения, который ведет от блока входа информации в поток к потенциально опасному блоку. Для определения наиболее перспективных векторов исследования нам необходимо выработать ряд правил, которые будут описывать искомый тип уязвимости. Например, если мы ищем уязвимости типа Buffer Overflow, то одним из правил может являться правило уязвимости функции «strcpy». Т.е., разрабатывая правило, мы, например, получаем следующий набор условий:

1) Потенциально опасный блок кода содержит инструкцию вызова функции «strcpy». Следует отметить, что некоторые компиляторы строят подобные вызовы методом, когда инструкция «call» указывает на некоторый адрес, который далее выполняет переход с помощью инструкции «jmp» на импортированную функцию и т.п. Кроме того, функции могут быть развернуты в сам исследуемый программный модуль как inline-функции.
2) Путь до потенциально опасного блока не содержит или содержит минимальное число вызовов других функций стандартной библиотеки, которые могут ограничить входные данные. В случае если путь не содержит, он может получить максимальный приоритет. В дальнейшем приоритет выставляется в соответствии с количеством вызовов функций, которые могут ограничить входные данные (чем их больше, тем меньше приоритет).

На основе полученных данных мы можем составить список возможных путей и их приоритетов. Следует отметить, что не каждый из предложенных путей может быть реальным. В некоторых случаях эвристический анализатор может предложить варианты переходов в рамках одного пути, которые, согласно алгоритму функционирования программного компонента, являются взаимоисключающими. В дальнейшем подобные пути можно

анализировать динамическими методами для повышения качества анализа и проверки предположений уязвимости.

Заключение

Таким образом, предложенный метод может позволить обнаружить некоторую часть уязвимостей в автоматизированном режиме. Точность обнаружения напрямую зависит от качества составления правил и их соответствия типу искомых уязвимостей. Каждый отдельный тип уязвимости характеризуется определенными признаками, которые должны быть отражены в правилах. При составлении правил необходимо учитывать, что как недостаточно точное, так и чересчур подробное описание типа уязвимости может негативно сказаться на качестве анализа, поскольку в первом случае мы будем иметь слишком большое число ложных срабатываний системы, а во втором – можем пропустить большое количество уязвимостей.

Литература

[1] http://www.busybox.net/

[2] www.sysinternals.com/

[3] http://msdn.microsoft.com/en-us/library/windows/hardware/gg463279.aspx

[4] http://msdn.microsoft.com/en-us/library/windows/hardware/gg463294.aspx

[5] Russinovich. M. – Windows Internals Part-1, 6-th Edition – 2012

[6] http://ru.wikipedia.org/wiki/Избирательное_управление_доступом

[7] http://ru.wikipedia.org/wiki/Упрелвение_доступом_на_основе_ролей

[8] http://ru.wikipedia.org/wiki/Мандатное_управление_доступом

[9] http://en.wikipedia.org/wiki/Anonymous_(group)

[10] http://en.wikipedia.org/wiki/LulzSec

[11] http://en.wikipedia.org/wiki/Stuxnet

[12] http://en.wikipedia.org/wiki/Flame_(malware)

[13] http://support.drweb.com/questions/mac/

[14] http://linuxgid.ru/servery-s-kodom-yadra-linux-zarazili-troyanom/

[15] http://www.securelist.com

[16] www.viva64.com

[17] http://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml