Анализ кода и информационная безопасность

Лекция 03

Ошибка типа «переполнение буфера»

МГУ / ВМК / СП

0301

ПЕРЕПОЛНЕНИЕ БУФЕРА · ВЫПОЛНЕНИЕ ПРОИЗВОЛЬНОГО КОДА НА ИСПОЛНИМОМ СТЕКЕ

CWE-118

- **118** — Improper Access of Indexable Resource (‘Range Error’)
  - **119** — Improper Restriction of Operations within the Bounds of a Memory Buffer
    - **788** — Access of Memory Location After End of Buffer
      - **126** — Buffer Over-read
      - **122** — Heap-based Buffer Overflow
      - **121** — Stack-based Buffer Overflow
    - **786** — Access of Memory Location Before Start of Buffer
      - **127** — Buffer Under-read
      - **124** — Buffer Underwrite (‘Buffer Underflow’)

CWE-121

- Программа осуществляет запись в буфер, размещенный на стеке, по неверному индексу, превышающему наибольший допустимый.
- Ошибка типична для языков Си и Си++, а также для ассемблера.
- Возможные последствия эксплуатации уязвимости:
  - нарушение доступности — аварийное завершение или зависание программы;
  - нарушение конфиденциальности, целостности и доступности — перехват потока управления, внедрение произвольного кода.
- CWE оценивает вероятность эксплуатации как очень высокую.

Переполнение буфера: пример 1

```
#include <string.h>

#define BUFSIZE 16

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

    strcpy(buf, argv[1]);
    return 0;
}
```

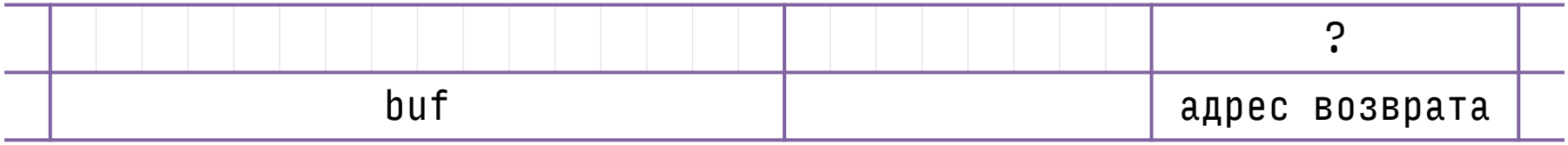
?

argv[1] = “0123456789ABCDEFXXXXXXXXYYYYYYYY”

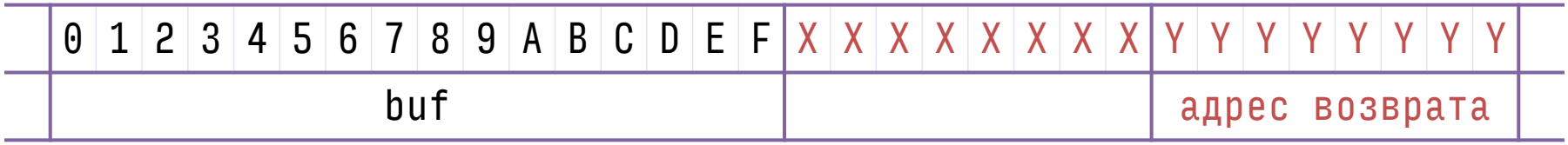
Переполнение буфера: пример 1

```
004003F0:        48 83 EC 18             SUB    RSP, 0x18
004003F4:        48 8B 76 08             MOV    RSI, QWORD [RSI + 8] ; argv[1]
004003F8:        48 89 E7                MOV    RDI, RSP ; buf
004003FB:  <1>   E8 C0 FF FF FF          CALL   strcpy
00400400:  <2>   31 C0                   XOR    EAX, EAX
00400402:        48 83 C4 18             ADD    RSP, 0x18
00400406:        C3                      RET
```

Точка <1>: перед вызовом strcpy



Точка <2>: после вызова strcpy



Переполнение буфера: пример 1

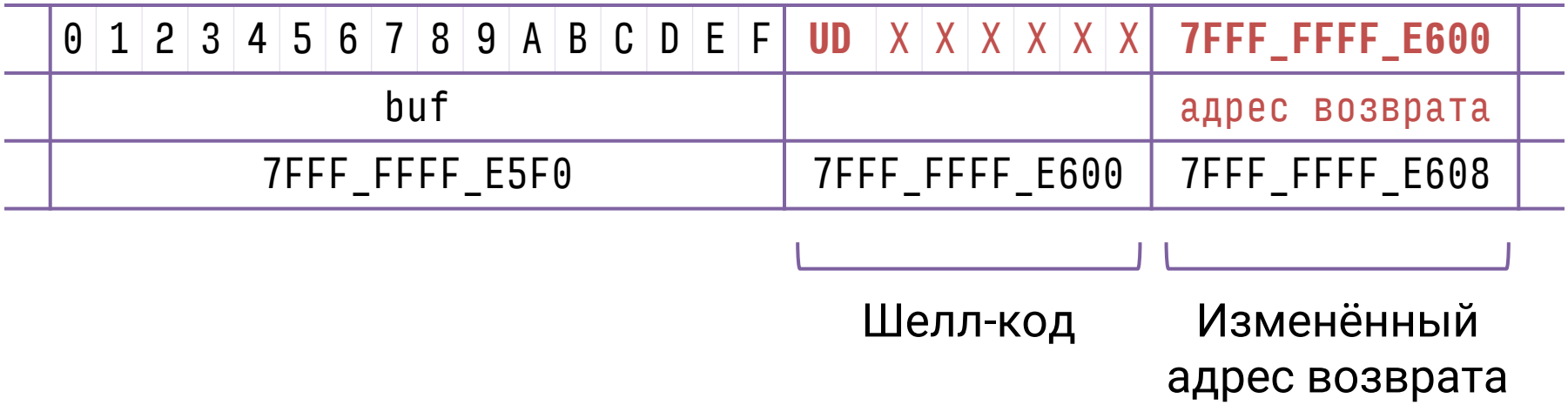
```
icee@debian:~/CAISCourse/L03/CodeSnippets$ ./030101 0123456789ABCDEFXXXXXXXXYYYYYYYY
Segmentation fault

icee@debian:~/CAISCourse/L03/CodeSnippets$ gdb -q --args ./030101 0123456789ABCDEFXX
XXXXXXYYYYYYYY
Reading symbols from ./030101...done.
(gdb) run
Starting program: /home/icee/CAISCourse/L03/CodeSnippets/030101 0123456789ABCDEFXXXX
XXXXYYYYYYYY

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400406 in main (argc=<optimized out>, argv=<optimized out>)
    at 030101.c:12
12  }
```

Переполнение буфера: пример 1

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffe610:
 rip = 0x400406 in main (030101.c:12); saved rip = 0x5959595959595959
 source language c.
 Arglist at 0x7fffffffe600, args: argc=<optimized out>, argv=<optimized out>
 Locals at 0x7fffffffe600, Previous frame's sp is 0x7fffffffe610
 Saved registers:
  rip at 0x7fffffffe608
(gdb) print $rsp
$1 = (void *) 0x7fffffffe608
```

0 1 2 3 4 5 6 7 8 9 A B C D E F	X X X X X X X X	Y Y Y Y Y Y Y Y
buf		адрес возврата
7FFF_FFFF_E5F0	7FFF_FFFF_E600	7FFF_FFFF_E608

Переполнение буфера: пример 1



Переполнение буфера: пример 1

	0123456789ABCDEF	0F 0B X X X X X X	00 E6 FF FF FF 7F 00 ??	
	buf		адрес возврата	
	7FFF_FFFF_E5F0	7FFF_FFFF_E600	7FFF_FFFF_E608	

Проблемы

1. `7FFF_FFFF_E608`: байт не может иметь значение `00` из-за использования `strcpy` — копирование прерывается при достижении нуль-терминатора.
2. `7FFF_FFFF_E60F`: значение байта в общем случае неизвестно, не может быть переписано.

Переполнение буфера: пример 1

	0123456789ABCDEF	X	0F	0B	X	X	X	X	X	01	E6	FF	FF	FF	7F	00	00	
	buf									адрес возврата								
	7FFF_FFFF_E5F0	7FFF_FFFF_E600								7FFF_FFFF_E608								

Решение проблем

1. 7FFF_FFFF_E608: разместим шелл-код со следующего байта.
2. 7FFF_FFFF_E60F: старший байт адреса возврата — нулевой в соответствии со стандартной картой памяти процесса в Linux.

Переполнение буфера: пример 1

```
icee@debian:~/CAISCourse/L03/CodeSnippets$ gdb -q --args ./030101 `echo -ne "0123456
789ABCDEFX\x0F\x0BXXXXX\x01\xE6\xFF\xFF\xFF\x7F"`
Reading symbols from ./030101...done.
(gdb) run
Starting program: /home/icee/CAISCourse/L03/CodeSnippets/030101 0123456789ABCDEFX
XXXXX◊◊◊◊

Program received signal SIGILL, Illegal instruction.
0x00007fffffffe601 in ?? ()
```

Замечание

При запуске программы вне отладчика такое поведение может не воспроизводиться, т.к. адреса в стеке могут отличаться.

Подготовка эксплоита

1. Анализ исходного кода программы, обнаружение ошибки переполнения буфера.
2. Анализ бинарного кода программы, его разметка в соответствии с исходным.
3. Подготовка карты стека на момент переполнения:
   - расположение переменной `buf`;
   - расположение сохранённого адреса возврата.
4. Проработка допустимого формата содержимого эксплоита:
   - шаблон с расположением фиксированных полей (поле адреса возврата);
   - ограничения на значения отдельных байтов.
5. Оформление окончательного вида эксплоита: конкретные значения байтов с внедрённым шелл-кодом.

Условия реализации атаки

1. Исполнимый стек: внедряемый код размещён на стеке. Если система не позволяет выполнять код из диапазона адресов, относящегося к стеку, атака не удастся.
2. Относительно корректное завершение функции: после того, как перезаписан адрес возврата и предшествующие ему значения в стеке, функция должна доработать до команды `ret`, чтобы выполнился внедряемый код.
3. Постоянные адреса: в рамках последовательных запусков программы адреса объектов в стеке не должны меняться, т.к. иначе перезаписанный адрес возврата перестанет быть корректным, и вместо перехвата потока управления можно будет получить лишь аварийное завершение программы.

Противодействие выполнению кода на стеке

Неисполнимый стек (W^X, DEP) — технология, позволяющая помечать сегменты или страницы памяти как неисполняемые. При попытке передать управление на код, размещённый в такой памяти, происходит аварийное завершение процесса.

Аппаратно поддерживается на уровне страничной трансляции во всех процессорах x86-64, а также в более новых x86 и ARM.

В более старых моделях x86 возможна более медленная и сопряжённая с дополнительными ограничениями на исполнимые файлы реализация с использованием сегментной трансляции.

«Канарейка» на стеке

Общая идея: проверять факт перезаписи стека непосредственно перед адресом возврата перед выходом из функции.

«Канарейка» — как правило случайное значение, которое размещается на стеке перед адресом возврата. Перед выходом из функции происходит сравнение значения в стеке с исходным значением. Если значения не совпадают, программа аварийно завершается.

«Канарейка» на стеке

Стек без «канарейки»

	?
локальные переменные	адрес возврата

Стек с «канарейкой»

	C	?
локальные переменные	«канарейка»	адрес возврата

Стек с испорченной «канарейкой» после переполнения

	XXXXXXXX != *C*	YYYYYYYY
локальные переменные	«мёртвая канарейка»	адрес возврата

Переполнение буфера: пример 2

```
00400460:       48 83 EC 28             SUB    RSP, 0x28
00400464:       48 8B 76 08             MOV    RSI, QWORD [RSI + 8]
00400468:       48 8D 7C 24 08          LEA    RDI, [RSP + 8]
0040046D:       64 48 8B 04 25 28 00    MOV    RAX, QWORD FS:[0x00000028]
                00 00
00400476:       48 89 44 24 18          MOV    QWORD [RSP + 0x18], RAX
0040047B:       31 C0                   XOR    EAX, EAX
0040047D:       E8 9E FF FF FF          CALL   strcpy
00400482:       48 8B 54 24 18          MOV    RDX, QWORD [RSP + 0x18]
00400487:       64 48 33 14 25 28 00    XOR    RDX, QWORD FS:[0x00000028]
                00 00
00400490:       74 05                   JE     0x00400497
00400492:       E8 99 FF FF FF          CALL   __stack_chk_fail
00400497:       31 C0                   XOR    EAX, EAX
00400499:       48 83 C4 28             ADD    RSP, 0x28
0040049D:       C3                      RET
```

Исходный код совпадает с примером 1, но компилируется с ключом `-fstack-protector`, который включает использование «канарейки».

Обход «канарейки» на стеке

1. «Канарейка» проверяется только перед выходом из функции, однако перехват потока управления может быть осуществлён раньше (перезаписываемый указатель на функцию).
2. «Канарейка» может быть перезаписана, если в функции есть ошибка CWE-123 (‘Write-what-where Condition’). Также в этом случае может быть перезаписан адрес возврата, а «канарейка» останется неизменной.
3. Если программа содержит, например, ошибку CWE-126 (‘Buffer Over-read’), или возможны множественные попытки (brute force), то значение «канарейки» может быть извлечено из стека, после чего возможна эксплуатация переполнения буфера с известным значением «канарейки»
4. Перехват обработчика исключения на Windows (SEH-эксплоит).

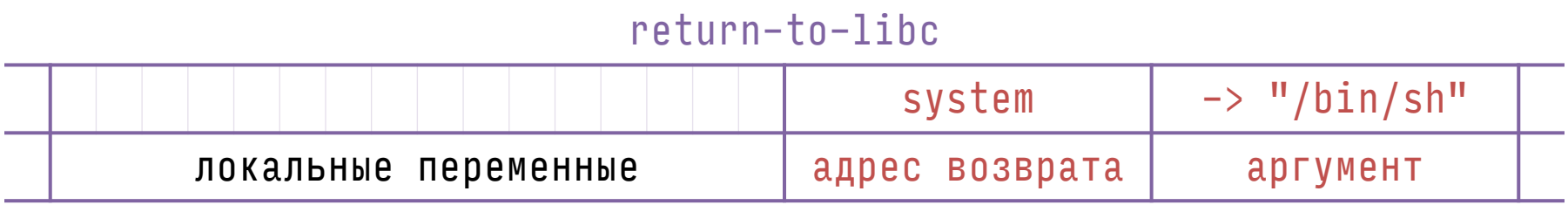
0302

ВЫПОЛНЕНИЕ ПРОИЗВОЛЬНОГО КОДА НА НЕИСПОЛНИМОМ СТЕКЕ

Обход DEP: return-to-libc

Вместо передачи управления на код внутри буфера можно заменить адрес возврата на адрес известной библиотечной функции, например `system` из стандартной библиотеки Си.

```
#include <stdlib.h>

int
system(const char *command);
```



Противодействие return-to-libc

```
icee@debian:~/CAISCourse/L03/CodeSnippets$ sudo cat /proc/self/maps | egrep "(heap|stack|libc)"
    01375000-    01396000 rw-p 00000000 00:00    0 [heap]
7fc6205c6000-7fc62075d000 r-xp 00000000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7fc62075d000-7fc62095d000 ---p 00197000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7fc62095d000-7fc620961000 r--p 00197000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7fc620961000-7fc620963000 rw-p 0019b000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7ffc40229000-7ffc4024a000 rw-p 00000000 00:00    0 [stack]

icee@debian:~/CAISCourse/L03/CodeSnippets$ sudo cat /proc/self/maps | egrep "(heap|stack|libc)”
    01661000-    01682000 rw-p 00000000 00:00    0 [heap]
7fa93f7c7000-7fa93f95e000 r-xp 00000000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7fa93f95e000-7fa93fb5e000 ---p 00197000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7fa93fb5e000-7fa93fb62000 r--p 00197000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7fa93fb62000-7fa93fb64000 rw-p 0019b000 08:01 4124 /lib/x86_64-linux-gnu/libc-2.23.so
7ffde81c3000-7ffde81e4000 rw-p 00000000 00:00    0 [stack]
```

Рандомизация адресного пространства (ASLR) — изменение карты памяти процесса при каждом запуске (процесса или системы).

Противодействие return-to-libc

```
icee@debian:~/CAISCourse/L03/CodeSnippets$ cat /proc/self/maps | grep cat
00400000-0040c000 r-xp 00000000 08:01 20725 /bin/cat
0060b000-0060c000 r--p 00000000 08:01 20725 /bin/cat
0060c000-0060d000 rw-p 00000000 08:01 20725 /bin/cat

icee@debian:~/CAISCourse/L03/CodeSnippets$ cat /proc/self/maps | grep cat
00400000-0040c000 r-xp 00000000 08:01 20725 /bin/cat
0060b000-0060c000 r--p 00000000 08:01 20725 /bin/cat
0060c000-0060d000 rw-p 00000000 08:01 20725 /bin/cat
```

- Без дополнительных действий (PIE) адрес загрузки основного исполняемого файла не рандомизируется.
- Можно попытаться обойти защиту, используя только постоянные адреса.
- Return-oriented programming (ROP).

ROP

Гаджет — последовательность команд в исполняемых секциях программы, заканчивающаяся командой RET.

Возвратно-ориентированное программирование (ROP) — техника построения экслоита, при которой полезная нагрузка формируется в виде цепочки гаджетов с известными постоянным адресами.

- Как из набора гаджетов «собрать» эксплоит?
- Где найти гаджеты?
- Какие гаджеты могут быть полезны?
- Как найти именно те гаджеты, которые нужны?

ROP

Запись числа в память: обычный способ

	v1		*v2*					
RSP +	0	8	16	24	32	40	48	56

```
MOV RAX, QWORD [RSP]
MOV RCX, QWORD [RSP + 16]
MOV QWORD [RCX], RAX
```

Запись числа в память: ROP

	v1	*g2*	*v2*	*g3*				
RSP +	0	8	16	24	32	40	48	56

1. Управление передаётся на *g1*.
2. Последовательно отрабатывает вся цепочка гаджетов: *g1*, *g2*, *g3*.

```
g1: POP RAX
    RET

g2: POP RCX
    RET

g3: MOV QWORD [RCX], RAX
    RET
```

Поверхность атаки

- Linux:
  - рандомизированный стек;
  - рандомизированная куча (mmap / brk);
  - рандомизированные адреса загрузки динамических библиотек;
  - рандомизированные адреса загрузки PIE-программ;
  - нерандомизированные адреса загрузки остальных программ.
- Windows:
  - рандомизированный стек;
  - рандомизированные кучи;
  - рандомизированные адреса служебных структур данных;
  - рандомизированные адреса загрузки динамических библиотек и программ;
  - по умолчанию рандомизация включена только для программ, которые её явно запрашивают (opt-in).

Варианты полезной нагрузки

1. Выполнить необходимые действия, полностью организовав их в виде цепочки гаджетов.
2. Разрешить выполнение кода на стеке, изменив права доступа к соответствующим страницам (mprotect, VirtualProtect).
3. “Stack pivoting”: изменить значение RSP, выбрав в качестве нового стека контролируемый регион в куче.

Автоматизация ROP: Q

- Статья: Q: Exploit Hardening Made Easy.

- Автоматическая генерация полезной нагрузки по:
  - исполняемым модулям с известными адресами загрузки;
  - описанию функциональности полезной нагрузки на DSL-языке QooL.

- Выделение гаджетов:
  - дизассемблирование со всех байтовых смещений;
  - рандомизированное тестирование семантики (фаззинг);
  - доказательство семантики (символьное исполнение);
  - получение базы гаджетов.

Выделяемые гаджеты

Q-Op	Семантика	Пример
JumpG(*t*)	RIP := *t*	JMP RAX
MoveRegG(*t1*, *t2*)	*t1* := *t2*	XCHG RAX, RBP; RET
LoadConstG(*t1*, *c*)	*t1* := *c*	POP RBP; RET
ArithmeticG(*t1*, *t2*, *t3*, ***op***)	*t1* := *t2* ***op*** *t3*	ADD RAX, RDX RET
LoadMemG(*t1*, *t2*, *c*)	*t1* := [*t2* + *c*]	MOV RAX, QWORD [RAX + 0x60] RET
StoreMemG(*t1*, *c*, *t2*)	[*t1* + *c*] := *t2*	MOV BYTE [RAX + 0x20], DL RET
ArithmeticLoadG(*t1*, *t2*, *c*, ***op***)	*t1* := *t1* ***op*** [*t2* + *c*]	ADD ECX, DWORD [RBX + 0x40] RET
ArithmeticStoreG(*t1*, *c*, *t2*, ***op***)	[*t1* + *c*] := [*t1* + *c*] ***op*** *t2*	ADD BYTE [RBP - 0x30], AL RET

Генерация полезной нагрузки

- Трансляция входной QooL-программы в последовательность Q-операций с локально минимальной длиной.
- Q-операции набираются из тех гаджетов, которые обнаружены в атакуемом коде.
- Проблемы:
  - побочные эффекты в гаджетах;
  - распределение регистров для исключения конфликтов.

Ограничения Q

- Набор гаджетов не Тьюринг-полон:
  - но это и не требуется, т.к. достаточно вызвать mprotect или system с нужными аргументами;
  - вручную подготовленный набор гаджетов из стандартной библиотеки Си Тьюринг-полон.
- Какова вероятность успешной атаки?
  - 80% вероятность вызова уже используемой в программе функции — начиная с объёма кода в 20 KiB;
  - 80% вероятность вызова произвольной внешней функции — начиная с объёма кода в 100 KiB.

Аудит передач управления

CFI (Control Flow Integrity) — средства, обеспечивающие проверку корректности потоков управления в программе.

kBouncer: Efficient and Transparent ROP Mitigation — статья.

kBouncer проверяет последние передачи управления перед выполнением системного вызова. Информация о передачах управления автоматически сохраняется процессором в специальных регистрах, сброс и чтение которых возможны только из привилегированного кода.

Литература к лекции

Основные источники

1. Brian Chess, Jacob West. Secure Programming with Static Analysis / Addison-Wesley Professional, 2007:
   - глава 6 — “Buffer Overflow”.
2. Aleph One. Smashing the Stack for Fun and Profit.
3. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy.
4. Vasilis Pappas. kBouncer: Efficient and Transparent ROP Mitigation.

Литература к лекции

Дополнительные источники

1. Benjamin Randazzo. Smashing the Stack, an Example from 2013.
2. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Ling. Jump-Oriented Programming: a New Class of Code-Reuse Attack.