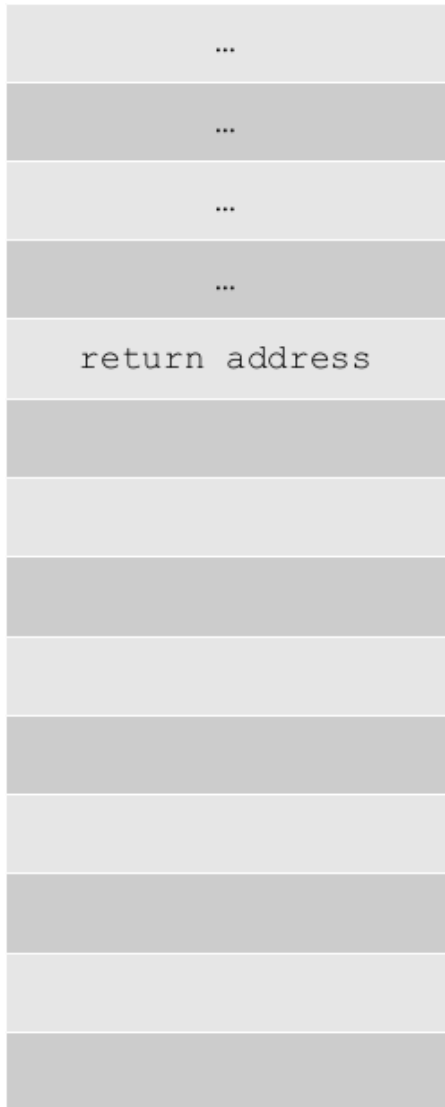
Buffer overrun attacks

Myrto Arapinis
School of Informatics
University of Edinburgh

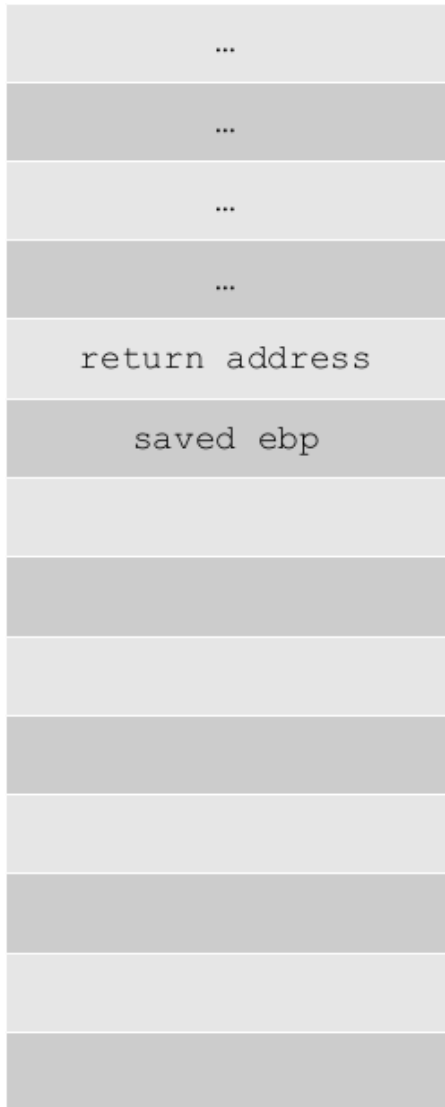
Basic buffer overflow (demo recap)

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

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
int x = read_stdi();
printf("x=%d\n", x);
}
```



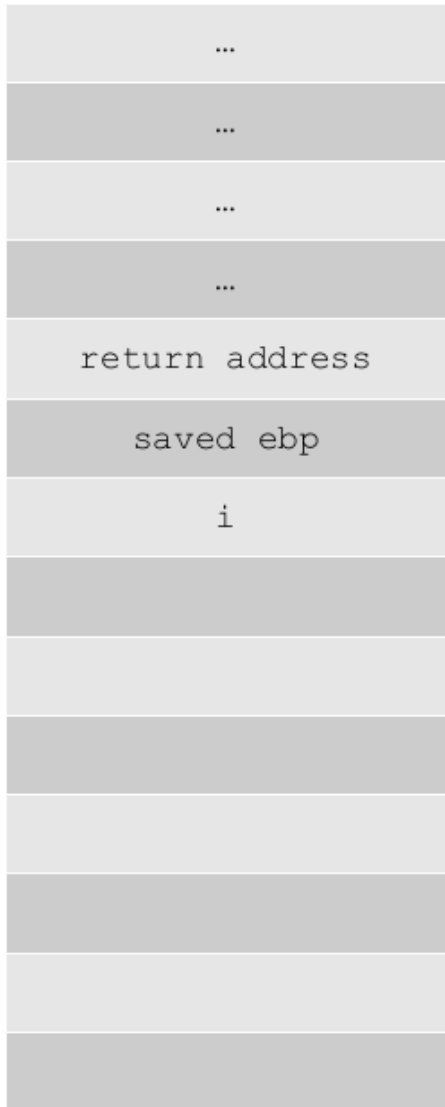
Basic buffer overflow (demo recap)

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

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
int x = read_stdi();
printf("x=%d\n", x);
}
```



Basic buffer overflow (demo recap)

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

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
int x = read_stdi();
printf("x=%d\n", x);
}
```



Basic buffer overflow (demo recap)

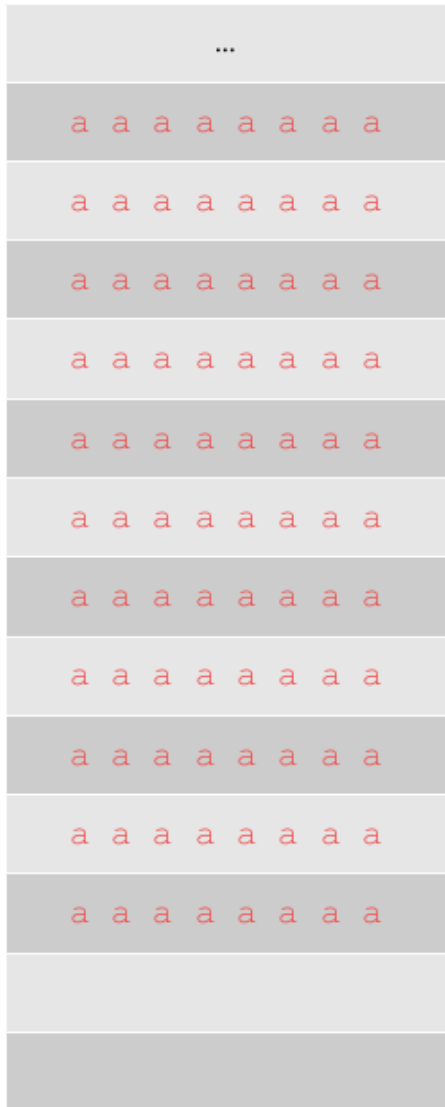
```
#include <stdio.h>
#include <stdlib.h>

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
int x = read_stdi();
printf("x=%d\n", x);
}
```

Basic buffer overflow (demo recap)

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

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
int x = read_stdi();
printf("x=%d\n", x);
}
```

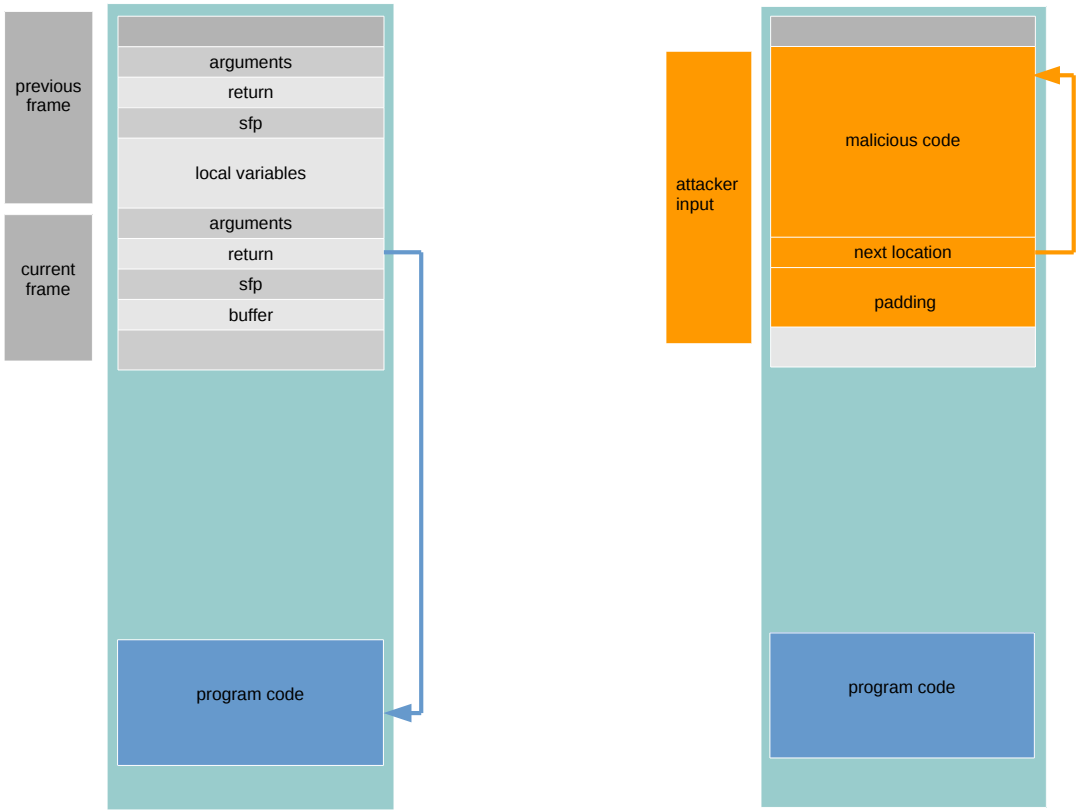
Basic buffer overflow (demo recap)

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

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
int x = read_stdi();
printf("x=%d\n", x);
}
```



Exploiting buffer overruns

Control hijacking



A buffer overflow can change the flow of execution of the program:

- load malicious code into memory
- make %eip point to it

Shellcode injection

Goal: "spawn a shell" - will give the attacker general access to the system

```
#include stdio.h
void main() {
char *name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

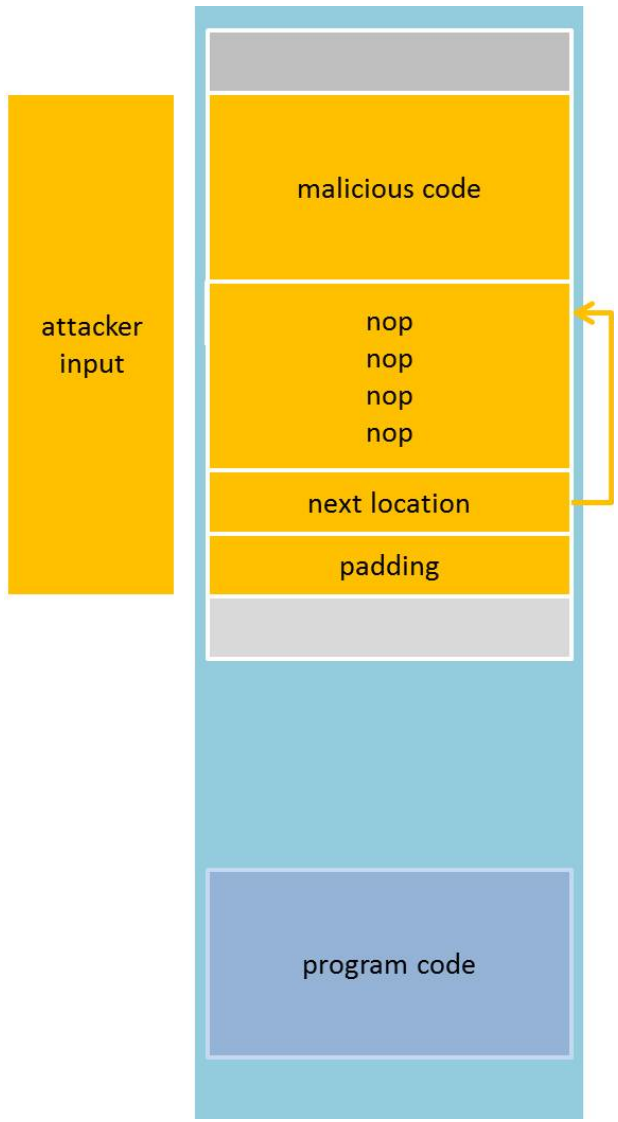
Machine code
(part of attacker's input)

- must inject the machine code instructions (code ready to run)
- the code cannot contain any zero bytes (`printf`, `gets`, `strcpy` will stop copying)
- can't use the loader (we're injecting)

The return address

Challenge: find the address of the injected malicious code?

- If code accessible: we know how far is the overflowed variable from the saved %ebp
- If code not accessible: try different possibilities!
  In a 32 bits memory space, there are 2^{32} possibilities
- NOP sled
  - guess approximate stack state when the function is called
  - insert many NOPs before Shell Code



First exploits of BOs

"All the News That's Fit to Print"

The New York Times

Late Edition

New York: Today, partly sunny, milder. High 58-64. Tonight, mostly cloudy. Low 48-54. Tomorrow, cloudy, windy, rain developing. High 57-63. Yesterday: High 56, low 41. Details, page D16.

VOL. CXXXVIII . . . No. 47,679 NEW YORK, FRIDAY, NOVEMBER 4, 1988 35 CENTS

Gov. Michael S. Dukakis having his picture taken by a 10-year-old fan at a town meeting in Fairless Hills, Pa., during a tour of the Northeast in which he emphasized the drug problem. Page A15. Vice Presi-

dent Bush addressed supporters a rally in Columbus, Ohio. Less than a week after Mr. Dukakis acknowledged being a liberal, Mr. Bush said yesterday that "this election is not about labels." Page A18.

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

Nationally, the percentage of eligible Americans who are registered is estimated to be 78.9 percent, down 2.2 points from the 1984 level.

The group's study overlooked that in many of the 30 states where final figures are available the decline was among

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an intrusion that raises questions about the vulnerability of the nation's computers, a Department of Defense network has been disrupted since Wednesday by a rapidly spreading "virus" program apparently introduced by a computer science student.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the nation and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry a tremendous variety of business and research information among military officials, researchers and corporations.

While some sensitive military data are involved, the computers handling the military asset management information, like that on the control of nuclear weapons, are thought not to have been touched by the virus.

Parallel to Biological Virus

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions in a computer, that is either placed on a floppy disk meant to be used with the computer or introduced when the computer is connected using ever telephone lines or data networks with other computers.

The programs can copy themselves into the computer's internal software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory. In this case, the virus program did nothing more than reproduce itself rapidly.

The program was apparently a result of an experiment, which

Continued on Page A21, Column 2

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.

Special to The New York Times

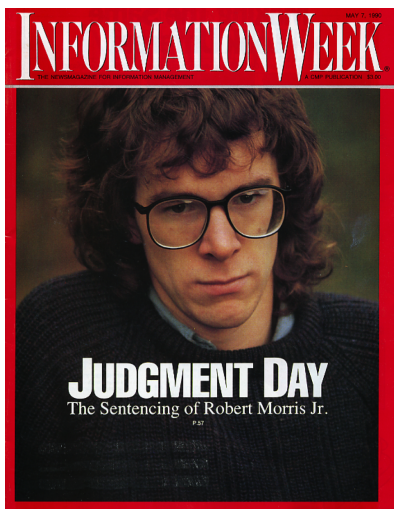
WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to ensure that the contractor does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this test is not met.

Broader Look at Consultants

The Justice Department's continuing criminal investigation has focused attention on consultants and the role in the congressional and military procurement and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own inves-



1988 - Robert Morris Jr.

First exploits of BOs

"All the News That's Fit to Print"

The New York Times

Late Edition

VOL. CXXXVIII . . . No. 47,679 NEW YORK, FRIDAY, NOVEMBER 4, 1988 35 CENTS

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

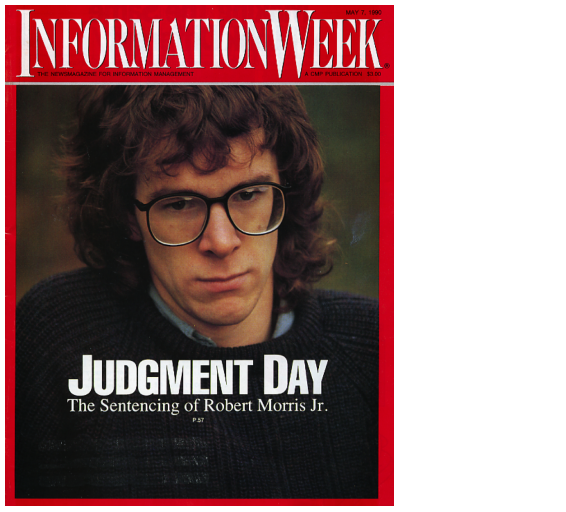
PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.

Registration Off Since 1984 Vote



1988 - Robert Morris Jr.

Title : Smashing The Stack For Fun And Profit

Author : Aleph1

```
                    .oO Phrack 49 Oo.

                  Volume Seven, Issue Forty-Nine

                       File 14 of 16

                  BugTraq, r00t, and Underground.Org
                        bring you

     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     Smashing The Stack For Fun And Profit
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                        by Aleph One
                    aleph1@underground.org

   `smash the stack` [C programming] n. On many C implementations
   it is possible to corrupt the execution stack by writing past
   the end of an array declared auto in a routine.  Code that does
   this is said to smash the stack, and can cause return from the
   routine to jump to a random address.  This can produce some of
   the most insidious data-dependent bugs known to mankind.
   Variants include trash the stack, scribble the stack, mangle
   the stack; the term mung the stack is not used, as this is
   never done intentionally. See spam; see also alias bug,
   fandango on core, memory leak, precedence lossage, overrun screw.


                          Introduction
                          ~~~~~~~~~~~~

   Over the last few months there has been a large increase of buffer
overflow vulnerabilities being both discovered and exploited.  Examples
of these are syslog, splitvt,
library, at, etc.  This paper
are, and how their exploits wo

   Basic knowledge of assembly
memory concepts, and experienc
We also assume we are working
system is Linux.

   Some basic definitions befo
block of computer memory that
type.  C programmers normally
commonly, character arrays.  A
declared either static or dyna
time on the data segment.  Dyn
the stack. To overflow is to f
We will concern ourselves only
known as stack-based buffer ov


Proc
~~~~
```

1996 - Elias Levy (Aleph One)

Buffer overrun opportunities

Unsafe libc functions

```
strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, ...)

...
```

Do not check bounds of buffers they manipulate!!

Use safe functions

```
int read_stdi(void){
   char buf[128];
   int i;
   fgets(buf, sizeof(buf), stdin);
   i = atoi(buf);
   return i;
}
```

But then...

... your program is as secure as its programmer is cautious. It is now up to the programmer to include all the necessary checks in his program :-/ and this is a tricky one...

Arithmetic overflows

- Limitation related to the representation of integers in memory
- In 32 bits architectures, signed integers are expresses in two's compliment notation
  - $0x00000000$ - $0x7fffffff$: positive numbers 0 - $(2^{31} - 1)$
  - $0x80000000$ - $0xffffffff$: negative numbers $(-2^{31} + 1)$ - (-1)
- In 32 bits architectures, unsigned integers are only positive numbers $0x00000000$ - $0xffffffff$.
  Once the highest unsigned integer is reached, the next sequential integer wraps around zero.

```
# include <stdio.h>
int main(void){
  unsigned int num = 0xffffffff;
  printf(``num + 1 = 0x%x\n'', num + 1);
  return 0;
}
```

The output of this program is: num + 1 = 0x0

Arithmetic overflow exploit (1)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf1, char *buf2,
                  unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 + len2) > 256){
    return -1;
  }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf);
  return 0;
}
```

Arithmetic overflow exploit (1)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf1, char *buf2,
                  unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 + len2) > 256){
    return -1;
  }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf);
  return 0;
}
```

Check can be bypassed by using suitable values for `len1` **and** `len2`, *e.g.* `len1 = 0x00000103`, `len2 = 0xfffffffc`, `len1+len2 = 0x000000ff` (decimal 255)

Arithmetic overflow exploit (2)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf, int len){
  char mybuf[256];
  if(len > 256){
    return -1;
  }
  memcpy(mybuf, buf, len);
  return 0;
}
```

Arithmetic overflow exploit (2)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf, int len){
  char mybuf[256];
  if(len > 256){
    return -1;
  }
  memcpy(mybuf, buf, len);
  return 0;
}
```

```
memcpy(void *s1, const void *s2, size_t n); // size_t
is unsigned
```

Check can be bypassed by using suitable values for `len`, *e.g.* `len = -1 = 0xffffffff` , will be interpreted as an unsigned integer encoding the value $2^{32} - 1$!

Arithmetic overflow exploit (3)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

Arithmetic overflow exploit (3)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

```
int myfunction(int *array, int len){
  int *myarray, i;
  myarray = malloc(len * sizeof(int));
  if(myarray == NULL){
    return -1;
  }
  for(i = 0; i < len; i++){
    myarray[i] = array[i];
  }
  return myarray;
}
```

Arithmetic overflow exploit (3)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

```
int myfunction(int *array, int len){
  int *myarray, i;
  myarray = malloc(len * sizeof(int));
  if(myarray == NULL){
    return -1;
  }
  for(i = 0; i < len; i++){
    myarray[i] = array[i];
  }
  return myarray;
}
```

Can allocate a size 0 buffer for `myarray` by using suitable value for `len`: `len = 1073741824` , `sizeof(int) = 4`, `len*sizeof(int) = 0`

The Ariane 5 Disaster

[Blexim] Basic Integer Overflows `http://phrack.org/issues/60/10.html#article`

Attempt to store a value in an integer which is greater than the maximum value the integer can hold
$\longrightarrow$ the value will be truncated

Ariane 5 rocket launch explosion due to integer overflow

Format strings

[Ref] scut/team teso. Exploiting Format String Vulnerabilities

- A format function takes a variable number of arguments, from which one is the so called format string

  Examples: `fprintf`, `printf`, `...`, `syslog`, `...`

```
printf(``The amount is %d pounds\n'', amnt);
```

Format strings

[Ref] scut/team teso. Exploiting Format String Vulnerabilities

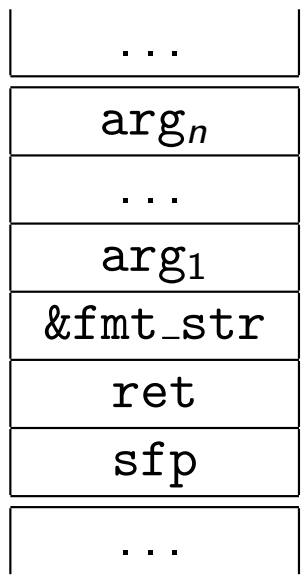
- A format function takes a variable number of arguments, from which one is the so called format string

  Examples: `fprintf`, `printf`, ..., `syslog`, ...

  ```
  printf(``The amount is %d pounds\n'', amnt);
  ```

- The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack

  Example: `printf(fmt_str,` $\texttt{arg}_1$`, ...,` $\texttt{arg}_n$`);`



Exploiting format strings

- If an attacker is able to provide the format string to a format function, a format string vulnerability is present

```
int vulnerable_print(char *user) {
  printf(user);
}

int safe_print(char *user){
  printf ("%s", user);
}
```

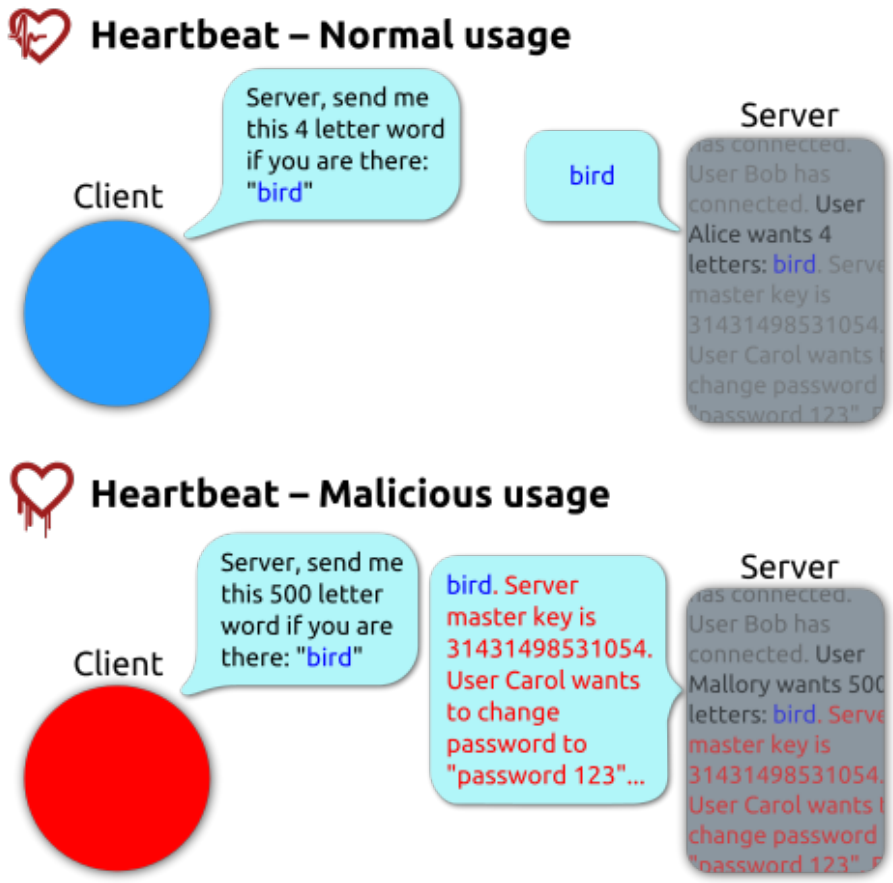
Format strings exploits

- We can view the stack memory at any location
  - walk up stack until target pointer found
  - `printf (‘‘%08x.%08x.%08x.%08x.%08x|%s|’’);`
  - A vulnerable program could leak information such as passwords, sessions, or crypto keys
- We can write to any memory location
  - `printf(‘‘hello %n’’, &temp)` – writes '6' into `temp`
  - `printf(‘‘hello%08x.%08x.%08x.%08x.%n’’)`

More buffer overflow opportunities

- Exception handlers
- Function pointers
- Double free
- ...

TLS Heartbleed



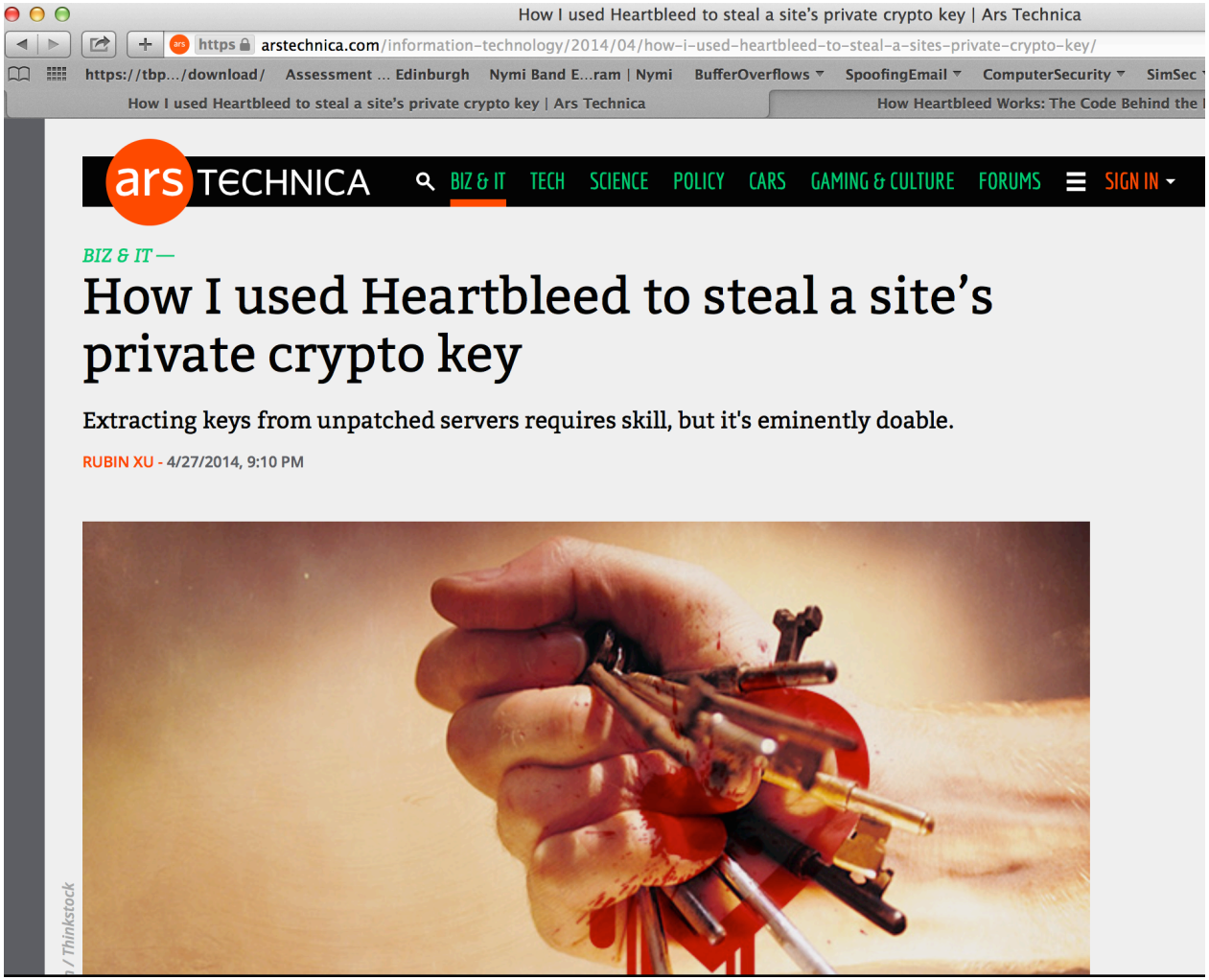
TLS Heartbleed

Then, OpenSSL will uncomplainingly copy 65535 bytes from your request packet, even though you didn't send across that many bytes:

```
1  /* Allocate memory for the response, size is 1 byte
2   * message type, plus 2 bytes payload length, plus
3   * payload, plus padding
4   */
5  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
6  bp = buffer;
7
8  /* Enter response type, length and copy payload */
9  *bp++ = TLS1_HB_RESPONSE;
10 s2n(payload, bp);
11 memcpy(bp, pl, payload);
12 bp += payload;
13 /* Random padding */
14 RAND_pseudo_bytes(bp, padding);
15
16 r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload +
```

That means OpenSSL runs off the end of your data and scoops up whatever else is next to it in memory at the other end of the connection, for a potential data leakage of approximately 64KB each time you send a malformed heartbeat request.

TLS Heartbleed



One of the most common attacks on memory safety

The CWE Top 25

Below is a brief listing of the weaknesses in the 2020 CWE Top 25, including the overall score of each.

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53
[18]	CWE-522	Insufficiently Protected Credentials	5.49
[19]	CWE-611	Improper Restriction of XML External Entity Reference	5.33
[20]	CWE-798	Use of Hard-coded Credentials	5.19
[21]	CWE-502	Deserialization of Untrusted Data	4.93
[22]	CWE-269	Improper Privilege Management	4.87
[23]	CWE-400	Uncontrolled Resource Consumption	4.14
[24]	CWE-306	Missing Authentication for Critical Function	3.85
[25]	CWE-862	Missing Authorization	3.77