lecture

Lecture on

Buffer Overflows

Technical Foundation, Attacks and Countermeasures

Walter Kriha

Goals for today

What is a buffer overflow? Types and risks

Learn how buffer overlows attacks work

C-language specific weaknesses

The basics of machine architecture and language runtime needed to understand how a buffer overlow works

Trace a buffer overflow from c-code to assembly code to memory layout

Defensive measures (stack protection, libsafe, canaries, data execution protection, address space layout randomization)

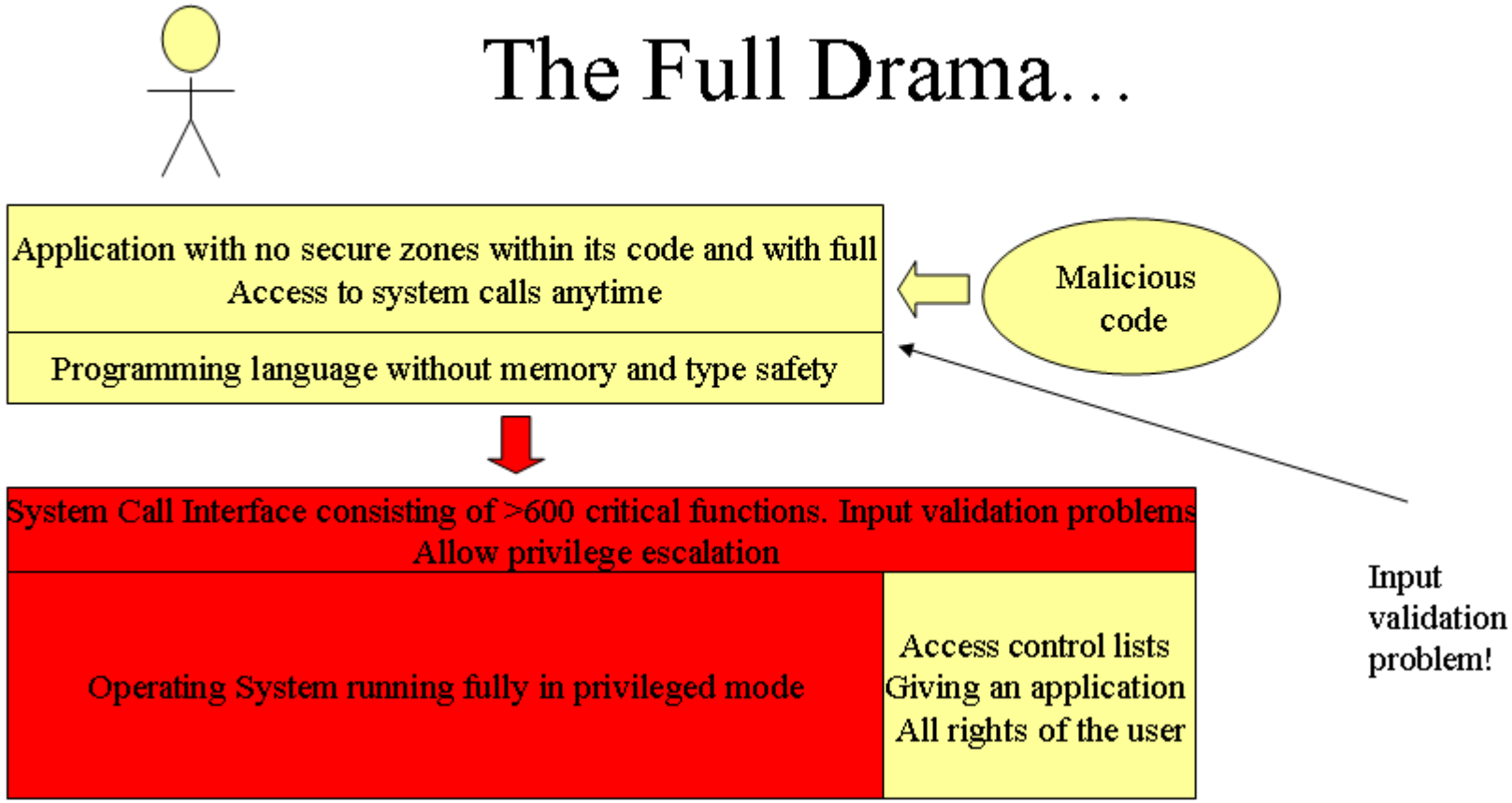
Are pictures safe? Is text harmless? What are the consequences for a security architecture? How can you prevent such attacks?

The importance of buffer overflows

Previously between 50% and 80% of all known vulnerabilities across operating systems were owed to buffer overflows. And while the focus of attackers has shifted to applications and their input validation problems buffer overflows are still a very dangerous vulnerability.

They are also hard to detect in both open source and binary distributed software – even though there is evidence that the open source process with hundreds to thousands of people scrutinizing code seems to have an edge here. But still: some code has been carefully audited and still buffer overflows were found later (e.g. bind). Manual inspections seems to be no cure and there is only one automated cure right now: do not use languages which are not memory safe (e.g. with unbounded buffers or unsafe type casts like C, C++)

The Full Drama…

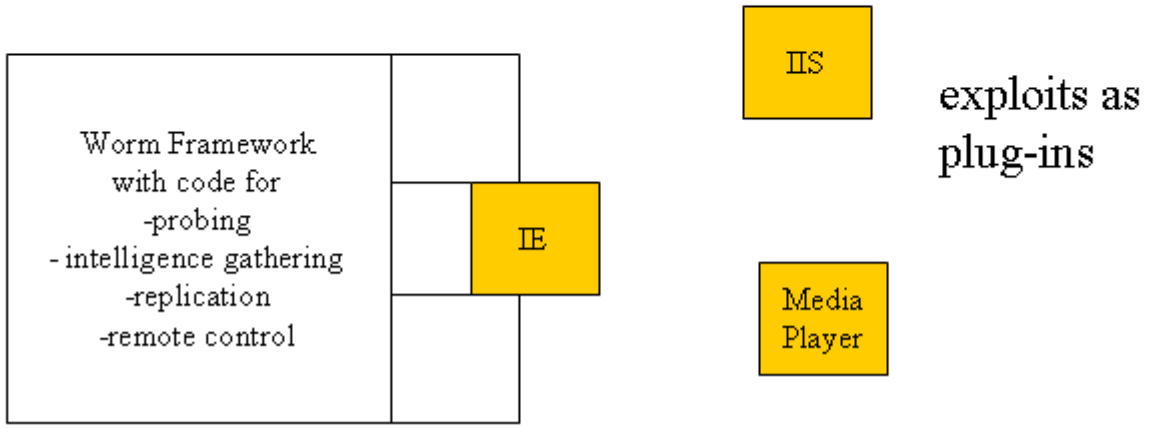


It is not just the fact that applications make errors in their input validation routines. Powerful user rights which are always applied in full or weaknesses in privileged code lead to complete takeovers with no damage restictions. A security analysis theoretically would have to look at the complete system in detail – something that is simply not possible.

Buffer Overflows from a security point of view

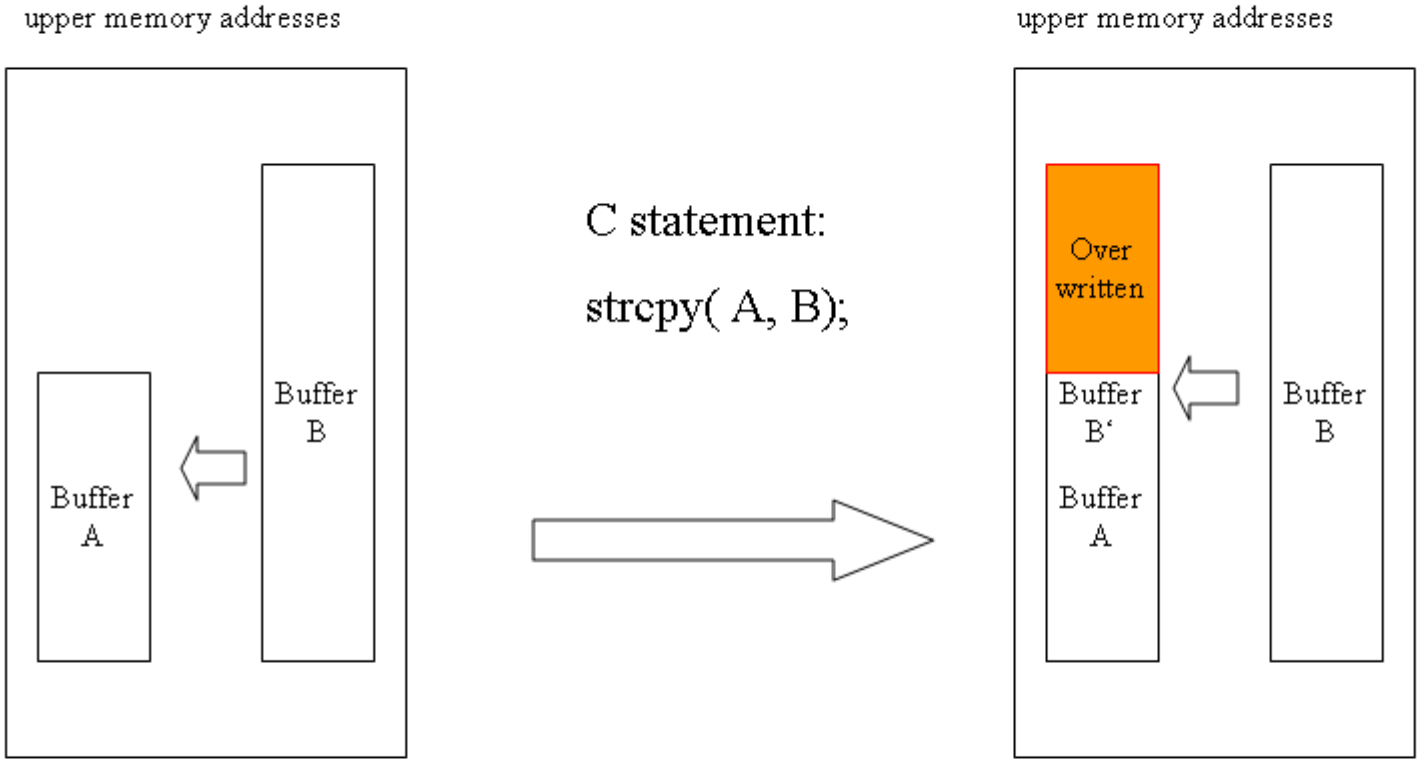
- In many cases they allow complete take-over of a machine by intruders
- Worm kits allow exploits within hours of invention.
- Attacks based on buffer overflows are hard to code (imagine what a person that is able to create a buffer overflow can do to your machine!)
- Such attacks can be packaged into easily usable attack tools allowing script kiddies to gain access to machines.
- Some prevention is possible using kernel and/or compiler technology.
- Today most systems rely on detection and updates (after the fact)
- Buffer overflows are deadly because of violations of the principle of least authority (POLA) in most operating systems.

Frameworks for Exploits



Modern framework technology separates everyday chores like finding proper systems, replicating the attach code and allowing remote control from the actual attack code on a specific program. If a new exploit is detected the frame for using it does already exist. This makes a „reaction within days“ promised by software vendors look much less attractive.

What is a buffer overflow technically?



The library function strcpy copies the source (Buffer B) over the destination (Buffer A) without recognizing that buffer A is smaller than B. The memory right behind buffer A will be overwritten with the content of buffer B. This may go unnoticed, can cause program malfunction OR be used for an attack on the system.

Example: Simple Buffer Overflow

```
#include <stdio.h>


int main(int argc, char** argv)  {
    int foo=0xeeee;
    char myArray[4];
    gets(myArray);
    printf(" print integer first: %x ", foo);
    printf("%s ", myArray);
}
```

```
Exception: STATUS_ACCESS_VIOLATION at eip=62626262
eax=00000029 ebx=00000000 ecx=00000029 edx=00000029
esi=00000000 edi=00402970
ebp=62626262 esp=0022FEE8
program=D:\walter\security\bufferoverflow\over.exe
cs=001B ds=0023 es=0023 fs=0038 gs=0000 ss=0023
Stack trace:
Frame     Function  Args
 531113 [main] over 928 handle_exceptions:
Exception: STATUS_ACCESS_VIOLATION
 545483 [main] over 928 handle_exceptions: Error
while dumping state (probably corrupted stack)
```

Compile and run this little program. Type in „a“ letters on the keyboard and type return. Start with one „a“ and watch the output. Then use more and more „a“ letters. What happens to the output? When does the program crash? What does the crashdump show? Look especially at the EIP and EBP registers? Do you see your „a“ letters somewhere? If you detect your input pattern in the registers of the CPU you know that you have found a way into the program. The rest is mere work.

Example: Simple Buffer Overflow (2)

```
#include <stdio.h>

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

    int foo=0xeeee;
    char myArray[4];
    gets(myArray);
    printf(" print integer first: %x ", foo);
    printf("%s ", myArray);

}
```

Keyboard Input (with return)	Display Output
a	Eeee a
aa	Eeee aa
aaa	Eeee aaa
aaaa	Ee00 aaaa
aaaaaaaaaaaaa	Core dump with EIP = 6161616161616161 (Hex 61 == `a`)

Our „aaaaaaaa..“ input from keyboard is now the address where the next instruction should be read by the CPU. Now we know how to point the CPU to code we placed on the stack

```
Exception: STATUS_ACCESS_VIOLATION at eip=61616161
eax=00000012 ebx=00000004 ecx=610E3038 edx=00000000 esi=004010AE
edi=610E21A0
ebp=61616161 esp=0022EF08
program=D:\kriha\security\bufferoverflow\over.exe, pid 720, thread main
cs=001B ds=0023 es=0023 fs=003B gs=0000 ss=0023
Stack trace:
Frame     Function  Args
  90087 [main] over 720 handle_exceptions: Exception:
STATUS_ACCESS_VIOLATION
 104452 [main] over 720 handle_exceptions: Error while dumping state
(probably corrupted stack)
```

A program crash is a way into the system!

Stack Layout

Function Parameter
Leftmost Function Parameter
RETURN Address
Caller BP copy
Foo
myArray[3]
myArray[1]
myArray[1]
myArray[0]

Address overwritten!

a
a
a
a

Gets() starts writing here

Keyboard Input (with return)	Stack layout
a	eeee a (first array element)
aa	eeee aa (first and second)
aaa	eeee aaa (first, second and third)
aaaa	ee00 aaaa (4 array elements + zero)
aaaaaaaaaaaaa	aaaaaaaaaaa (all local variables and the return address overwritten, crash on function return

The kernal trap interface

your code wants to send a message msg to stdout:

```
push  len      ;message length
 push   msg      ;message to write
 push   1        ;file descriptor (stdout)
  mov     AX, 0x4        ;system call number (sys_write)
 int 0x80         ; kernel interrupt (trap)
add     SP, 12          ;clean stack (3 arguments * 4)
push    0        ;exit code
mov     AX, 0x1         ;system call number (sys_exit)
 int 0x80       ; kernel interrupt we do not return from sys_exit there's no need to clean stack
```

The trap (system call interface) ist very important for attack code because it is POSITION INDEPENDENT! Your code is NOT LINKED with the running program and therefore does not know where specific library functions etc. are located in your program. The kernel interface is always just there and can be used to load Dynamic Link Libraries into the program.

Types of Buffer Overflows

- Heap/BSS overflows
- Stack overflows
- Data (re-)placing overflows
- Instructions (re-)placing overflows

A rough distinction can be made according to WHERE the overflow happens (in different memory regions e.g. heap, stack, data) and WHAT gets (over)written: data, function pointers or instructions. Instructions can only be replaced when they are not located in a read-only memory page.

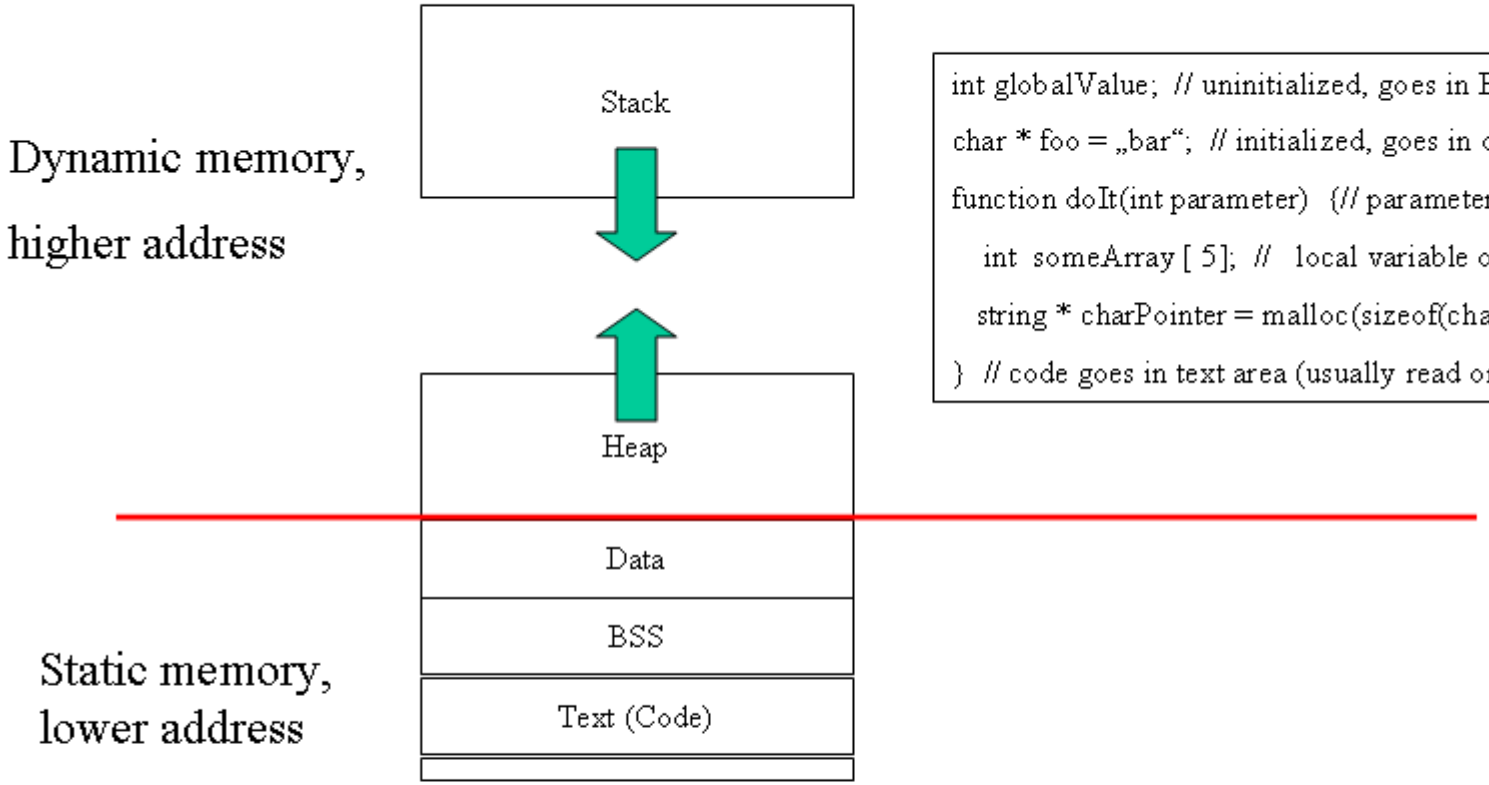
Attack locations

- Return address
- (Old) base pointer
- Function pointer as (local) variable
- Function pointer as (function) parameter
- longjmp buffer as (local) variable
- longjmp buffer as function param.

- Return address
- (Old) base pointer
- Function pointer as (local) variable
- Function pointer as parameter
- longjmp buffer as (local) variable
- longjmp buffer as function param.

From the iDefense paper on „a comparison of buffer overflow prevention...“ (see resources). The attacks are differentiated according to WHERE they happen (stack or heap/bss) and whether they are DIRECT overflows or overflow a pointer leading to the real target.

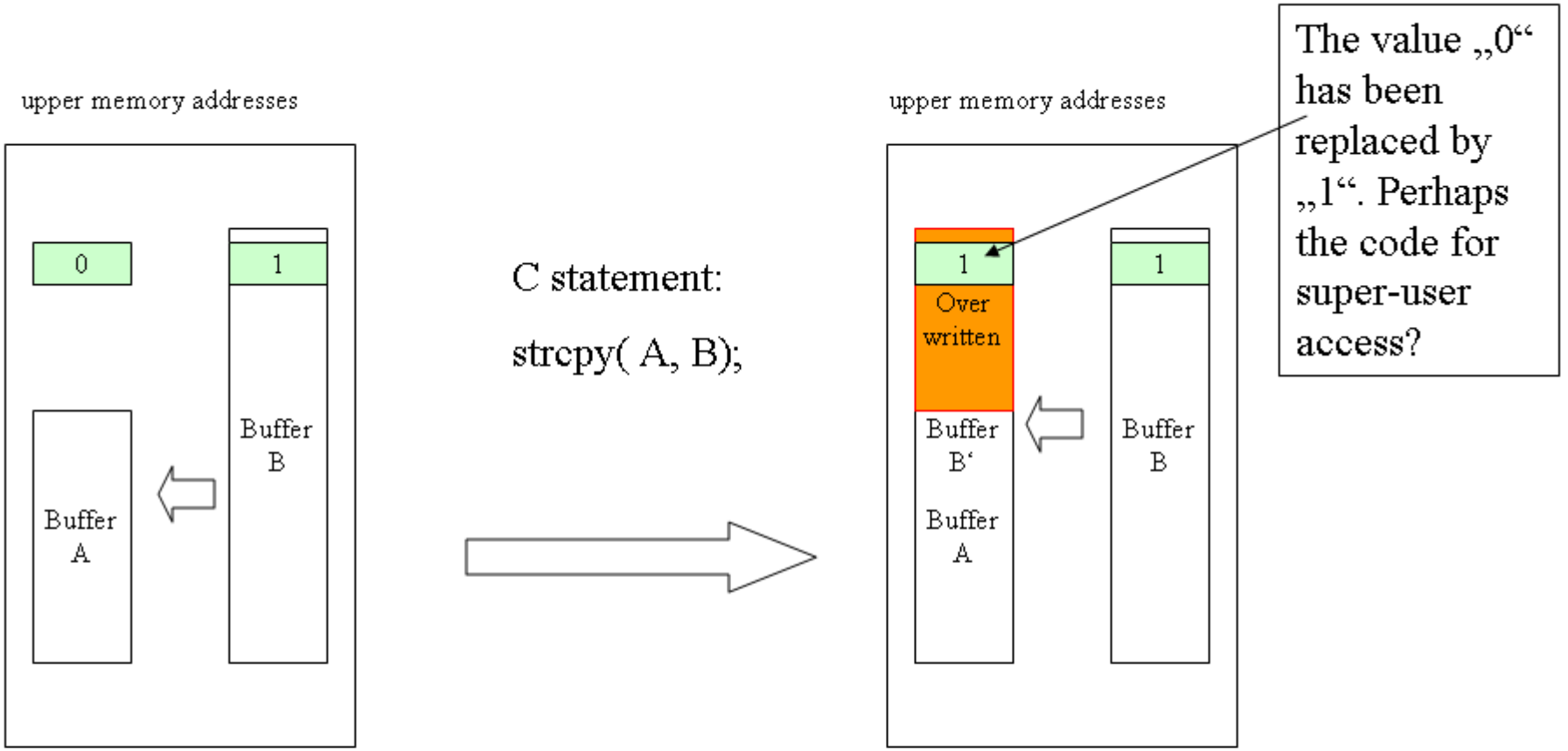
Program Memory Areas




```
int globalValue;  // uninitialized, goes in BSS
char * foo = „bar“;  // initialized, goes in data area
function doIt(int parameter)  {// parameter goes on stack
   int  someArray [ 5];  //   local variable on stack
  string * charPointer = malloc(sizeof(char)*100); // on Heap
}  // code goes in text area (usually read only at runtime)
```

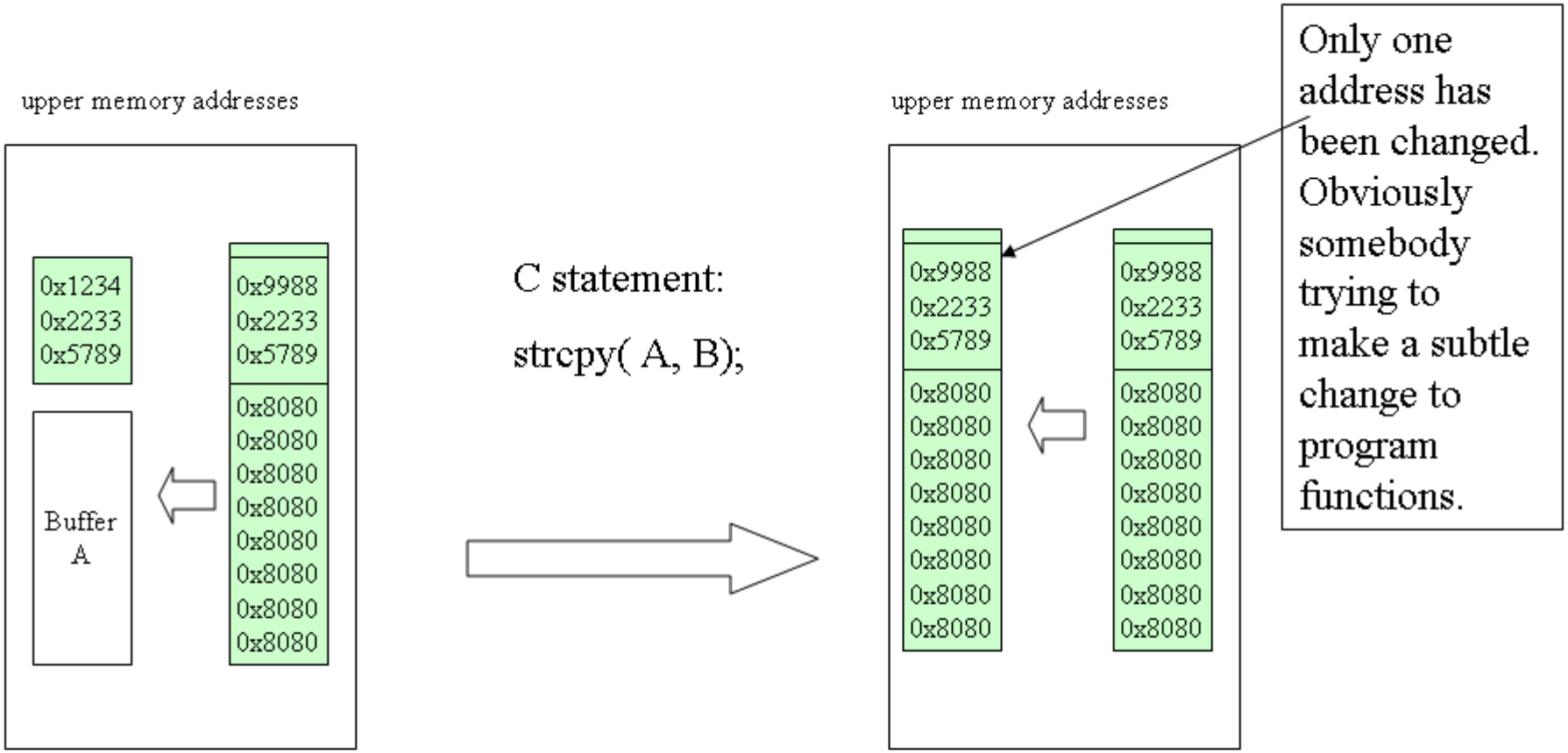
Please note that both stack and heap are dynamic. Heap grows upwards (caused by malloc() or new ClassXX() statements) and stack grows downwards. If the meet then you are in trouble. The tiny segment at the bottom of the virtual memory is called „zero page“ in some systems and serves to catch ininitialized pointer access. BSS content is usually initialized to 0 automatically at program start. Text is locked at program runtime to allow for pages being thrown out in low memory conditions. All other areas need to be stored in swap space if memory gets low because they may have changed during program runtime.

A data replacing overflow



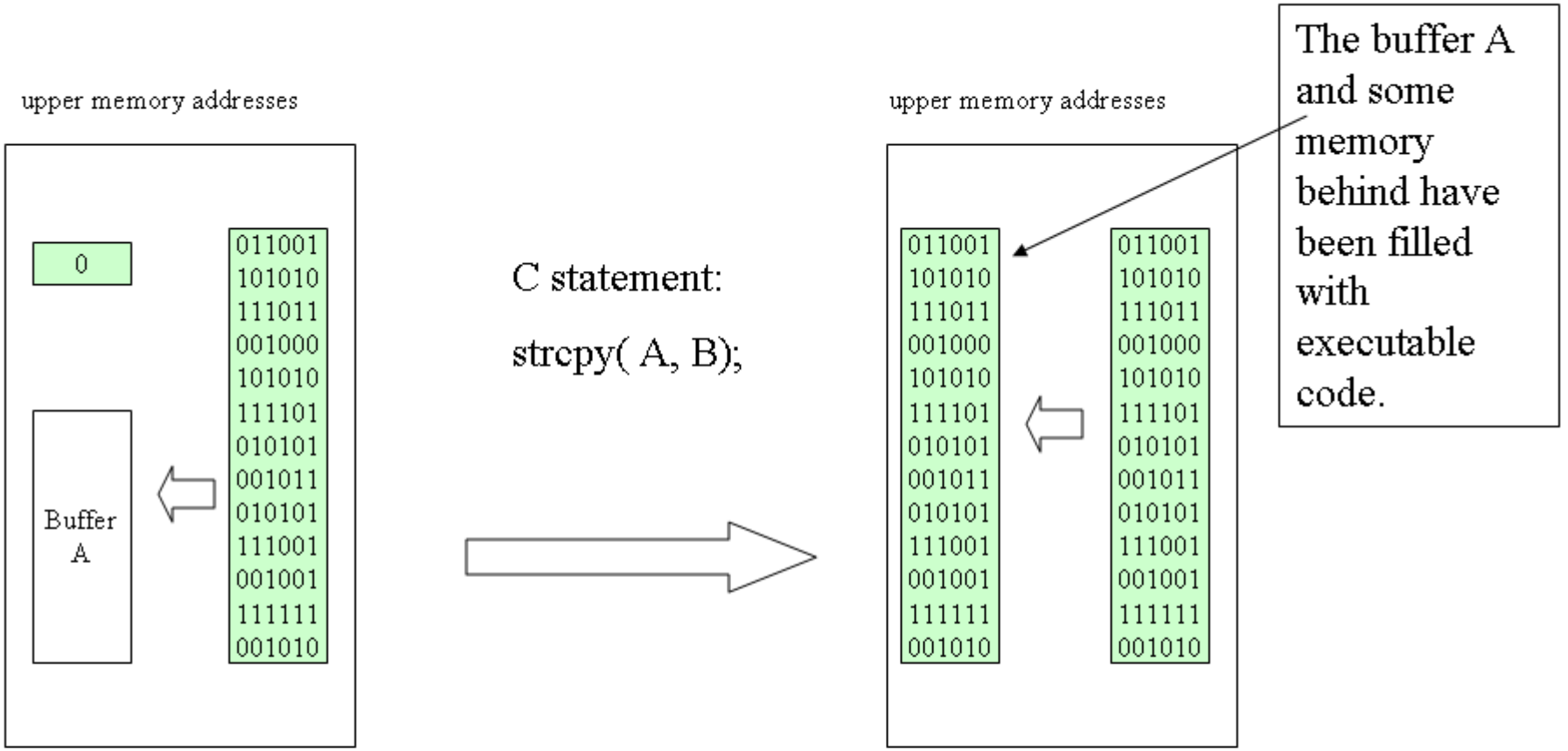
This kind of writing over data causes a program crash in most cases. A major problem for attacker is the fact that the hole between buffer A and the „0“ value needs to be overwritten as well. Like the data placing case this one needs no attack code to be started.

An address replacing overflow



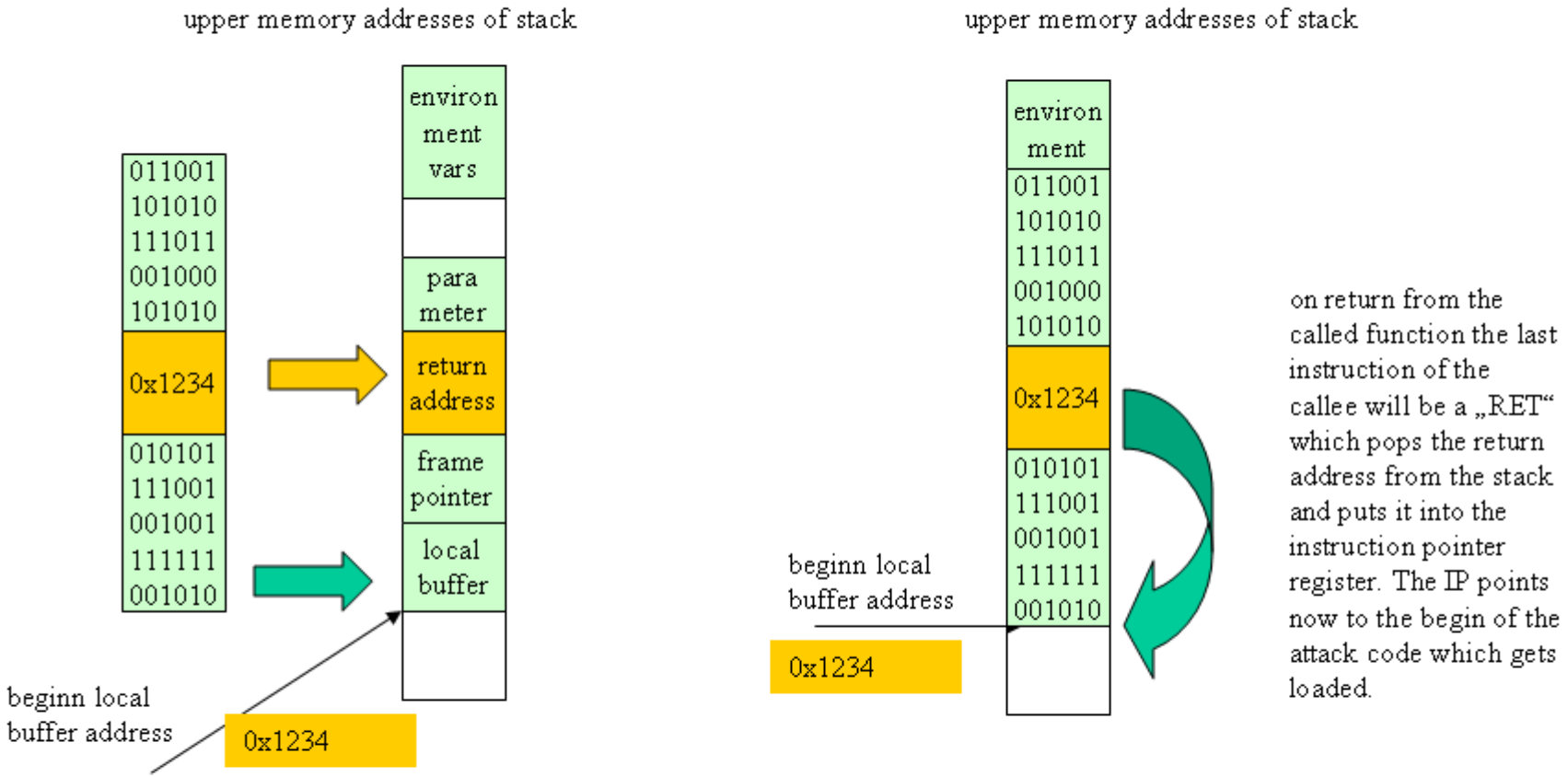
Filler bytes are used to reach the critical address area. The overwritten memory could hold a function jump table or the v-table of a c++ class holding the addresses of virtual methods.

A code-placing overflow



This overflow is typically not an „accident“ but the result of an attack. What is still missing is a way to start that attack code. The code itself has been carefully crafted to avoid „0“ values in between which would cause the copy operation to break. Sometimes even upper case or lower case values, newlines and carriage returns or byte values beyond 0x7f are a problem – depending on the copy operation as we will see later.

Stack smashing: place code and start it!



On the left side the attack code gets copied into the local buffer on the stack. The code is taylored to overwrite the return address (yellow) with the address of the local buffer – which now contains the attack code. The attack code could also be placed behind the return address which would then be overwritten to point to a higher memory address. This is the case when the place in the local buffer is too small to carry the attack code.

The stack smashing attack in slow-motion!

1) CPU Registers
2) C language calling convention
3) Stack layout in detail
4) C strings and C copy/scan functions
5) Where does the attack code come from?
6) How does the attack code get in? C string functions etc.
7) How do you find those vulnerable buffers?
8) The ugly side of creating the attack code
9) stacks with nulls in the address: Finding helpers in the running program

We will discuss these problems on the next slides. You may be exposed to some machine language! Please note that the stack is such an interesting attack place for buffer overflow attacks because it COMBINES code placement with the ability to start that attack code on return from a funtion (via RET).

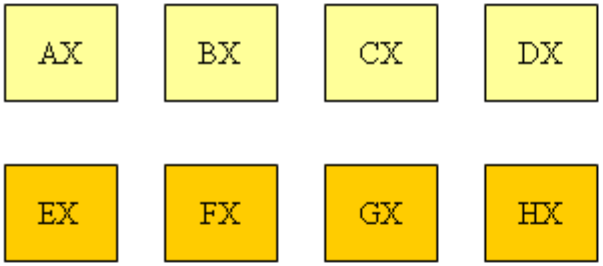
A very simple CPU

StackPointer (SP) points to the current stack address

BasePointer (BP) points to the current call frame address

Instruction Pointer (IP) points to the address of the current Instruction

Flag Register (Flags) hold the results of comparisons etc.



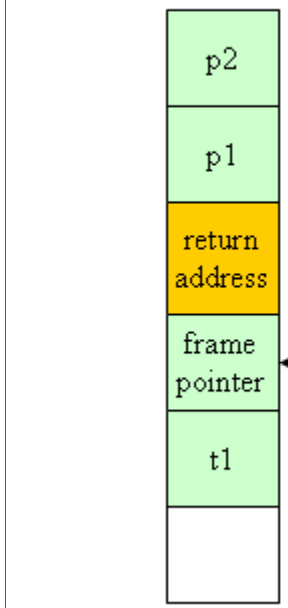
Eight general purpose registers. AX – DX are „scratch" registers – a callee can use them without saving them. If the caller has something valuable in those registers, it needs to store them some place (e.g. on the stack). EX-HX are callee's responsibility to save.

Our CPU is a 32 bit engine with little-endian byte order (like Intel and unlike Motorola). Unlike Intel the instruction set is fully orthogonal: all commands and addressing modes are available for all registers – no CS/DS index registers etc.

We run a small operating system on it where system calls use trap 0x80 (like linux) to transfer call, parameters and control to the kernel.

C-function calling convention

```
main(int argc, char** argv)  {
int i= 2;
int k=1;
int b;
b = add(i, k);
return address
}
int add (int p1, int p2) {
int t1 = p1 + p2;
return t1;
}
```



callee's BP points here

Caller:

push ax, bx, cx, dx (scratch registers) //optional

push p2 // push rightmost parameter on stack

push p1 // push left n parameter on stack

CALL _add // transfer control to „_add" address

// CALL pushes ++IP on stack (address of next instruction after return) and moved instruction pointer to _add location.

// on return from _add: stack cleanup

pop p1

pop p2

pop ax, bx, cx, dx (scratch registers, optional)

Callee:

push BP // save the callers frame pointer on stack

move SP, BP // put current Stack pointer in Base Pointer

move 1, --SP (put auto variable t1 on stack)

move (BP+2), ax // put first parameter in ax

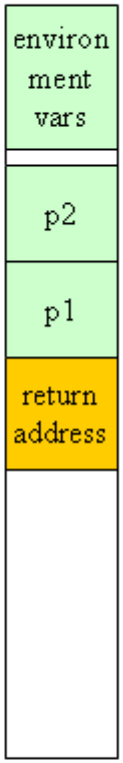
add (BP+3), ax // get second parameter, add to ax

incr SP // increment stack, get rid of local variable

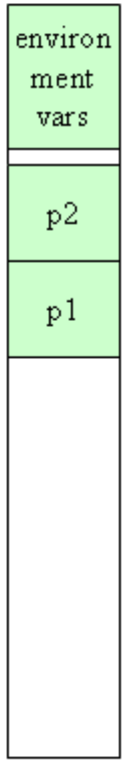
pop BP // restore callers Base Pointer

RET // ret pops return address into IP and jumps to it.

upper memory addresses of stack



after return to caller:



Calling convention rules:

- Caller saves „scratch registers“ if they store something valuable: These might be trashed by callee.
- Callee stores callers base pointer AND every non-scratch register IF changed by callee.
- Callee leaves the return value of the function in register ax where it will be picked up by caller.
- At return callee leaves stack exactly as it was when caller performed the call. BP is set to callers BP
- The Base Pointers (also called Frame Pointers) form a chain of pointers on the stack, each pointing to the previous callers Base Pointer.
- Caller will then also clean up the stack (remove parameters, restore scratch registers)

This is the so called „C-convention“. Others exist, e.g. Pascal convention where the callee removes callers parameters from stack. Another alternative is to pass parameters via memory registers or to omit the frame pointer. These special calling conventions require that both caller and callee have been compiled with the same optimizing compiler, otherwise they would not agree on register use etc.

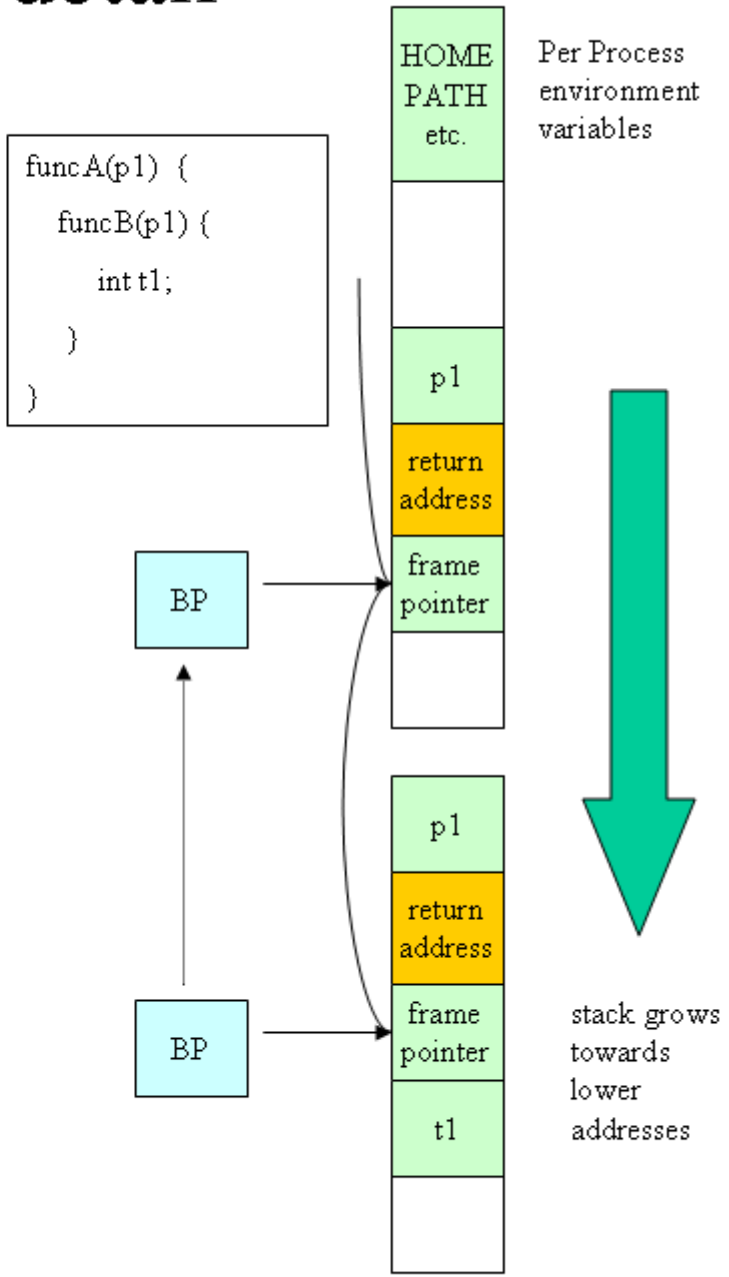
Stack layout in detail

At the bottom of the stack (highest address) are environment variables (put there by the process startup routine). After those a chain of stack frames is created by functions calling other functions. „main(argc, argv)“ is NOT the first function in a C-program. There are several startup and initialization functions which come first.

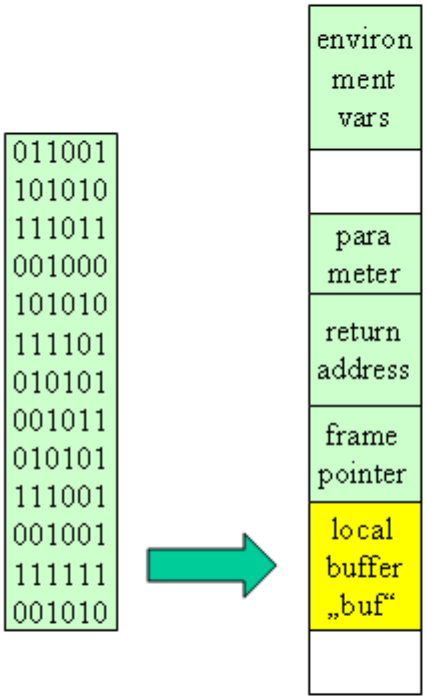
The stack is typically in a separate address range (provided that the System uses virtual memory addresses) than other program areas (e.g. code).

When functions call other functions and so on, the stack can grow considerably. The memory management system will automatically grow the stack towards lower addresses by allocating new memory pages.

Another reason why the stack grows is the use of automatic variables which are allocated on the stack. In this example it is only an integer but it could be a larger string array as well. The advantage of stack variables is that they are allocated much faster than heap variables and cleanup is easy: move current BP to SP before returning to caller and all automatic variables are gone (SP is now on stored BP)



Overflowing a stack buffer with C



C functions that do unbounded reads:

```
char buf[1024];
gets(buf);        // reads from console until a newline or EOF comes!!
```

C functions that do unbounded copy or format operations

```
void storeCharacters(char * inputChars)  {
char buf[1024];
strcpy(buf, inputChars);        // copies all input characters into buf, no matter how big inputChars really is.
```

Also tricky are formatted input functions like scanf:

```
char buf[1024];
sscanf(inputString, „%s“, &buf); // if inputString is bigger than buf an overflow occurs
```

There is a large number of C-function which does unbounded operations. For buffer overflow attacks the most interesting ones are those which manipulate strings or read from I/O channels. NEVER use an unbounded read function like gets. NEVER assume the size of input strings which have not been created within your own program. NEVER accept a format string for scanXXX functions from outside your program)

Strings and string copying in C

char * string = „abc“;

0x61|0x62|0x63|0

in memory representation always includes a trailing binary 0.

Strings in C are terminated by a binary „0“. This is how C-functions recognize the end of a string. There is no „String Class“ which would hold the length of a string in an extra attribute. In C asking for the length of a string means applying the „strlen()“ function which returns the length of the characters plus 1 for the trailing 0.

strcpy(stackbuffer, attackcode)

no 0 except last |0

An important consequence of the C-string handling for the attack code is that it cannot contain binary zeros because in that case the strcpy() function would terminate copying from source (attack code) to destination (stack buffer) at the first 0!

Safe and unsafe string functions in C

Danger!

- strcpy
- strcat
- sprintf
- scanf
- sscanf
- fscanf
- vfscanf
- vsprintf ….. (see resources)

Better:

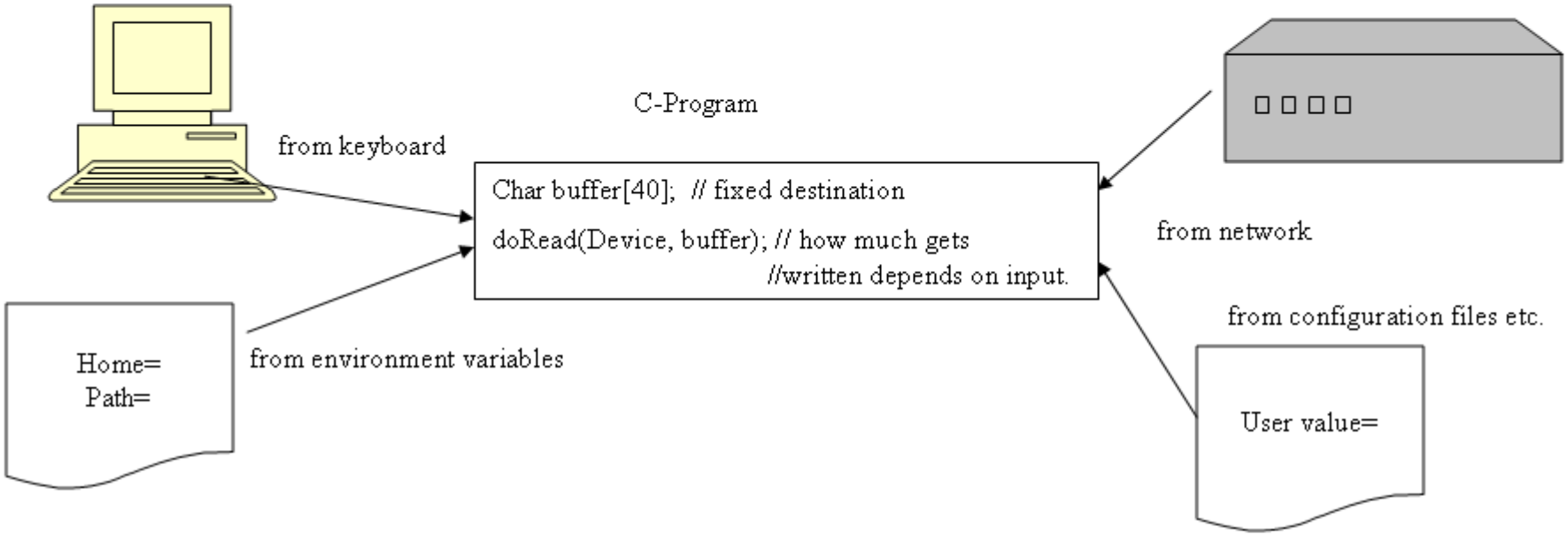
- All functions that take an additional length paramter to limit size: strncpy, strncat etc.
- Or use an additional precision specifier:

  sscanf(buf, „%255s, &buf)
  instead of

  sscanf(buf, „%s, &buf)

The general advice is to never make unbounded reads (e.g. gets()) or use external buffers without checking their size with strlen() before and making sure that your destination buffers are large enough to hold the copy. Question: This means you have to ALWAYS check if a buffer is filled externally (e.g. getting environment variables) AND that you have to check the SIZE with strlen(). The only difference to a generic bounds checking implementation of strings is that you can opt to NOT PERFORM THOSE CHECKS. A colleague later on may make a subtle change to the program and your assumptions about a buffer turn out to be wrong. This is very typcial of C-programs. We will see the same pattern with memory leaks later.

How does the attack code come in?



All this information comes from an external source into the program. If there is only a fixed buffer to receive that data, the size of the data needs to be checked to avoid a buffer overflow. Not even reading a programs own environment variables using getopt() etc., is safe because those variables may have been fixed to contain extremely large strings causing overflows. Or a users own default configuration file can be used to overflow buffers (Netmeeting example)

Creating buffer overflow attack code

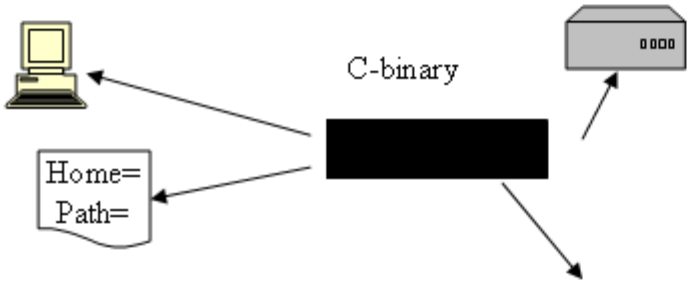
1) find vulnerable buffers
2) Try to find exact position of return address
3) Learn machine and OS architecture (e.g. memory areas)
4) Learn system call interface (traps) and basic functions.
5) Create new address to write over original return address
6) Create code for attack, load necessary library functions.
7) Allocate memory and download a remote control program (sub-seven, back-orifice etc.)

These are the basic steps to create attack code. This is actually quite difficult as we will see

Finding vulnerable buffers

Inspect source code

Only possible with open source software. This kind of software usually goes through many hands and buffer overflow vulnerabilities get fixed rather quickly or even before shipping.



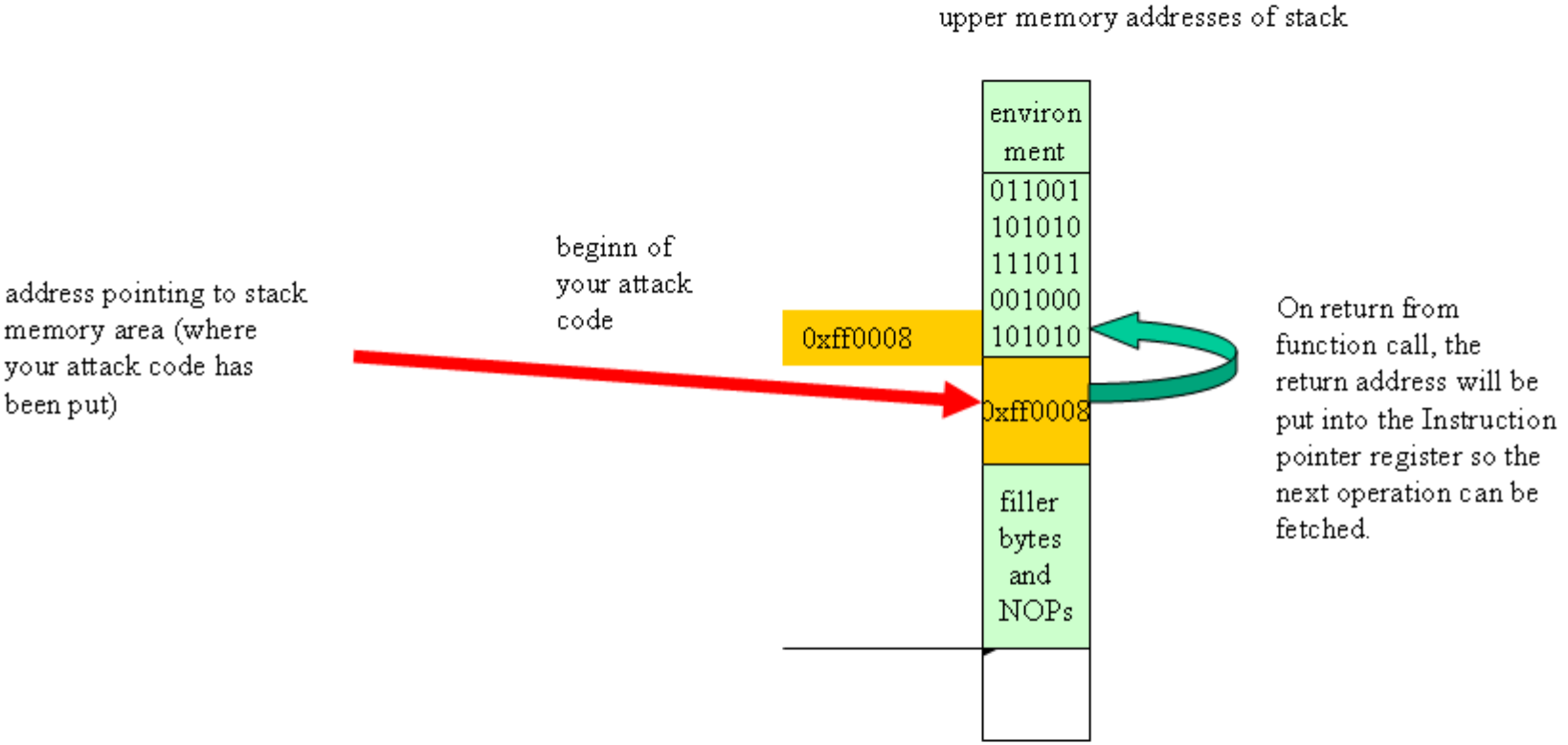
Play around with the black box

If you don't have source code you can either try to disassemble the program or start feeding it with all kinds of nonsense strings with regular patterns.

value= 0x8080808080808080808080808080808080808080808080808080808080808080

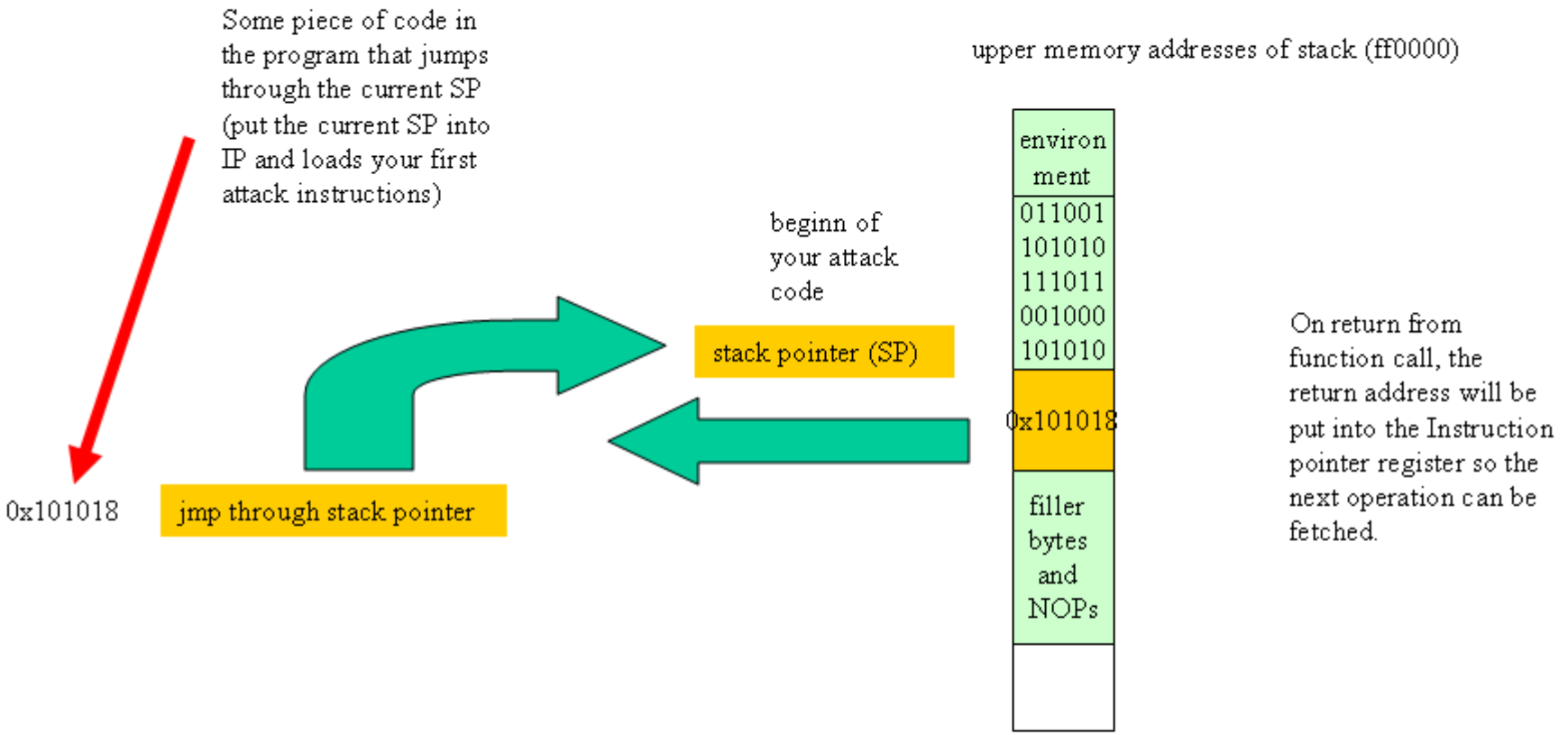
If the program crashes after reading the oversized input you have found a first hint at a possible vulnerability. Look at error messages. If you find an Instruction Pointer containing your pattern then you know that you have overwritten a return address! Bingo! Now you need to find the position of the return address by making your dummy string smaller until the program does no longer crash: now you are below the return address and possibly also the base pointer.

Create a stack address pointing to your code



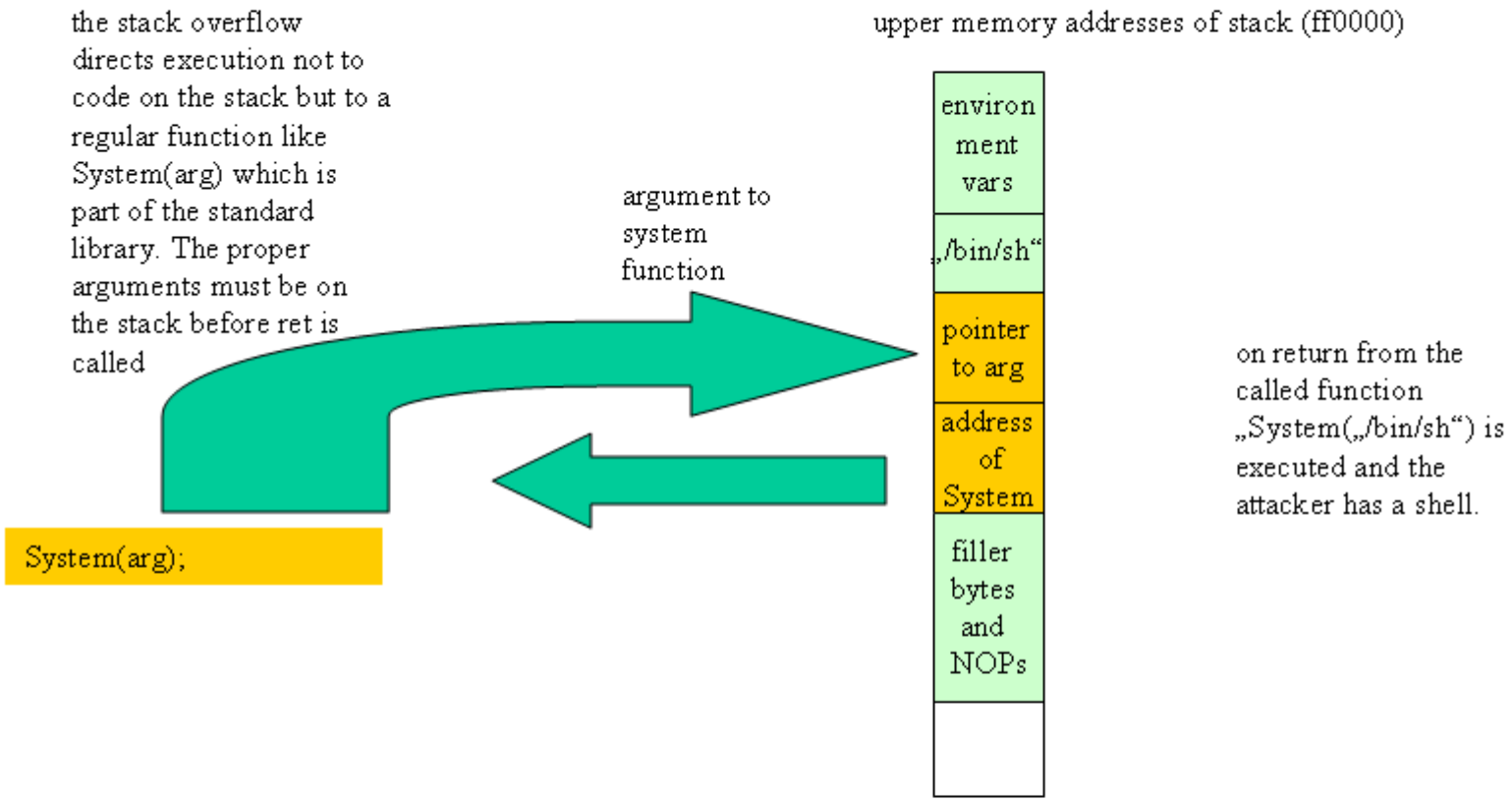
The problem here is that the stack segment (or area) may contain binary zeros! If a strcpy() function copies your attack code over the local stack buffer it would stop dead as soon as the first binary zero is hit – thinking that the string as at its end. Your code would not be copied completely. 0xff00 would be represented as 0x00ff in memory and therefore the last „ff“ would not be copied and the faked return address would be incomplete.

Use „helper“ in the attacked program



Because of the zeros in the stack address your attack code cannot overwrite the return address directly with the stack address of your code start location. But most programs contain little routines to jump through the stack. You just delegate the return to such a function which will jump to your first attack code instructions. Necessary zero values in your code can be simulated by XOR instructions (e.g. XOR AX, AX puts a 0 in AX)

Leverage return-to-libc mechanism



This attack is used if the stack segment is protected against execution of code.

Make calls from your attack code

1) Understand the system call trap interface
2) Find important library functions from DLL‘s
3) Load DLLs and functions needed by your attack code

The kernal trap interface

your code wants to send a message msg to stdout:

```
push  len      ;message length
 push   msg      ;message to write
 push   1        ;file descriptor (stdout)
  mov     AX, 0x4        ;system call number (sys_write)
 int 0x80         ; kernel interrupt (trap)
add     SP, 12          ;clean stack (3 arguments * 4)
push    0        ;exit code
mov     AX, 0x1         ;system call number (sys_exit)
 int 0x80       ; kernel interrupt we do not return from sys_exit there's no need to clean stack
```

The trap (system call interface) ist very important for attack code because it is POSITION INDEPENDENT! Your code is NOT LINKED with the running program and therefore does not know where specific library functions etc. are located in your program. The kernel interface is always just there and can be used to load Dynamic Link Libraries into the program.

Other interesting features of the C-language

- memory leaks
- pointer addressing
- resource chasing
- optimizations: omit-frame-pointer
- no byte code verifier. „Private“ etc. can be #undefined.
- bitwise const
- physical bindings to addresses

See: The Development of the C Language, Dennis M. Ritchie, Bell Labs/Lucent Technologies,

http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

Some last hints

- Use a decent debugger (e.g. gdb)
- Create assembly code from C with cc –S filename.c and inspect it.
- Use a good hex/octal viewer to understand byte ordering problems (e.g. od –t x1 filename will display the content of filename in hexadecimal. Use –t x2 to see the effects of byte swapping due to endianism)

Protection Mechanisms against Overflows

- Kernel based:
  - protect segments against executable code
  - randomly re-arrange program parts in memory to confuse address calculation in attack code
  - change stack to grow towards higher memory (makes arrays grow away from return address). HP computers do this.
- Compiler based:
  - insert magic values (canaries) and check their validity before returning
  - re-arrange variables, e.g. put arrays first after canary value

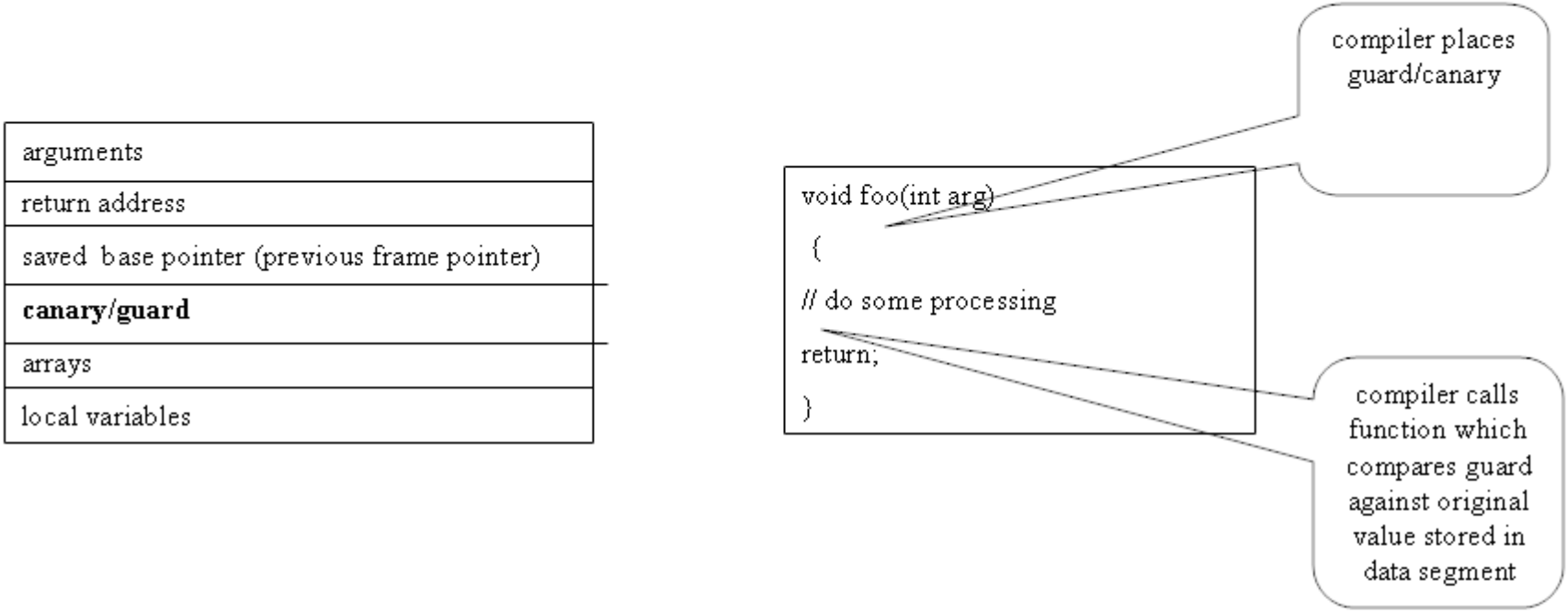
see the iDefense paper by Silberman/Johnson in resources.

Page Protection and Randomization

- Noexec:
  - mark stack and bss pages as non-executable
  - check mmap() calls to avoid creation of anonymous executable pages in memory (makes pages either executable OR writeable)
- ASLR: (address space layout randomization)
  - randomize location of kstack, ustack, mmap and elf binaries

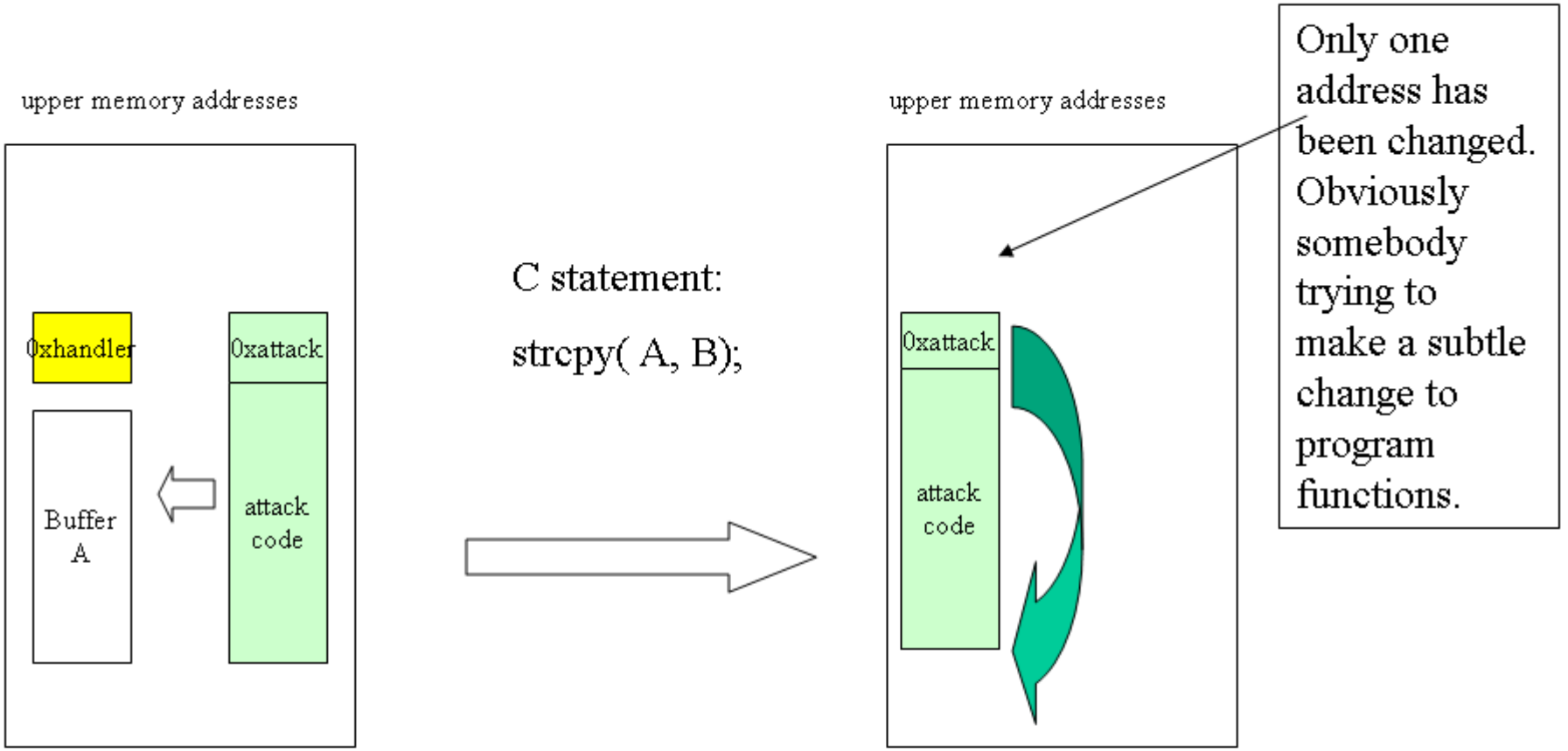
A lot of the difficulties come from deficiencies in the Intel x86 architecture which does not offer the proper protection bits. Instead, workaround with the TLB caching of data vs. code are used to lock pages against executable code. Some program code is broken by these measures, especially when code is generated at runtime.

Compiler Tricks with Canaries



Compiler protection tries to put all arrays right after the guard to avoid single variables (or function pointers) to be overwritten. The guard is checked against a save version stored somewhere else. Certain memory leak checkers also place markers behind arrays on the heap to detect overwrites. Besides using special compiler options for canaries one should ALWAYS use a good leak checker like purify to detect accidental overwrites by the program itself.

Exception handlers tricked



0xHandler is the address of an exception handler which should be called in case of an exception (e.g. canary overwritten). By replacing this address with our own address we can direct execution flow to wherever we want. This is one of various ways the windows stack protection in server 2003 was rendered useless. Creating an exception is easy. (again from iDefense...)

Vista Security Mechanisms: Problems

1. Heap and stack protection via canaris and variable re-ordering. Secure exceptions, secure list operations on heap: only when libraries are compiled with GS option. And only for some variable types. Attack code can still overwrite other variables on stack. 42% performance loss when GS option used! (This is more than memory safe languages cost!). Opt-in does not work.
2. Data Execution Protection (non-executable memory). Can cause serious compatibility problems and is therefore only sparingly used.
3. Adress Space Randomization. Many exceptions to randomization defined. Start addresses for a library on windows must be the same for all instances/applications! Many libraries cannot be randomized at all. Heap spraying allows malicious code to be loaded into executable regions (e.g. Java VM). Start addresses of libraries far from random.

Alexander Sotirov (VMWare)and Mark Dowd (IBM) at the Black Hat Conference in August 2008 „Bypassing Browser Memory Protections – setting back browser security by 10 years“

Combined Vista Security Attack

1. Use libraries which do not use heap/stack protection, which have executable regions and which restrict possible start addresses of other libraries considerably
2. Use the Internet Explorer‘s plug-in and extension mechanism as a dynamic loader for libraries. Use heap-spraying to load attack code
3. Use a known weakness to start the attack code

All the security exceptions and restrictions taken together undermine Vista security. No new techniques are needed.

Memory Attacks against VM‘s

Sudhakar Govindavajhala and Andrew W. Appel, Princeton University July 2003: Using Memory Errors to attack a Virtual Machine. Nice article on how to use (create) physical memory damages and how one bit is enough to break Virtual Machine type safety – and as a result of this memory safety. Arbitrary memory locations can be written once 2 different object types point to one memory location.
www.cs.princeton.edu/~sudhakar/talks/memerr.ppt and
http://www.cs.princeton.edu/~sudhakar/papers/memerr.pdf

The class definitions

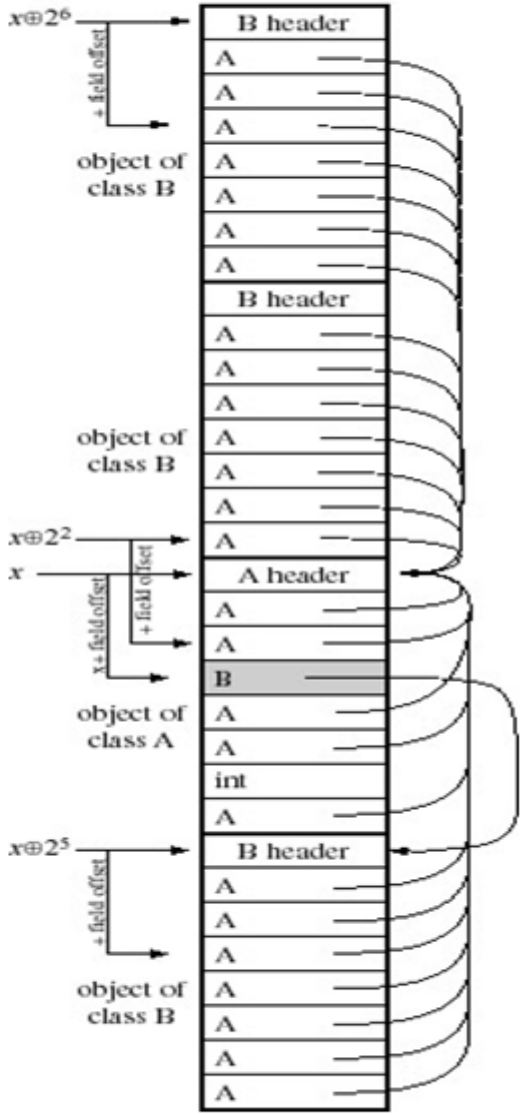
```
class A {
        A a1;
        A a2;
        B b;
        int i;
        A a5;
        A a6;
        A a7;
 }
// one instance
```

```
class B {
        A a1;
        A a2;
        A a3;
        A a4;
        A a5;
        A a6;
        A a7;
 }
// many instances
```

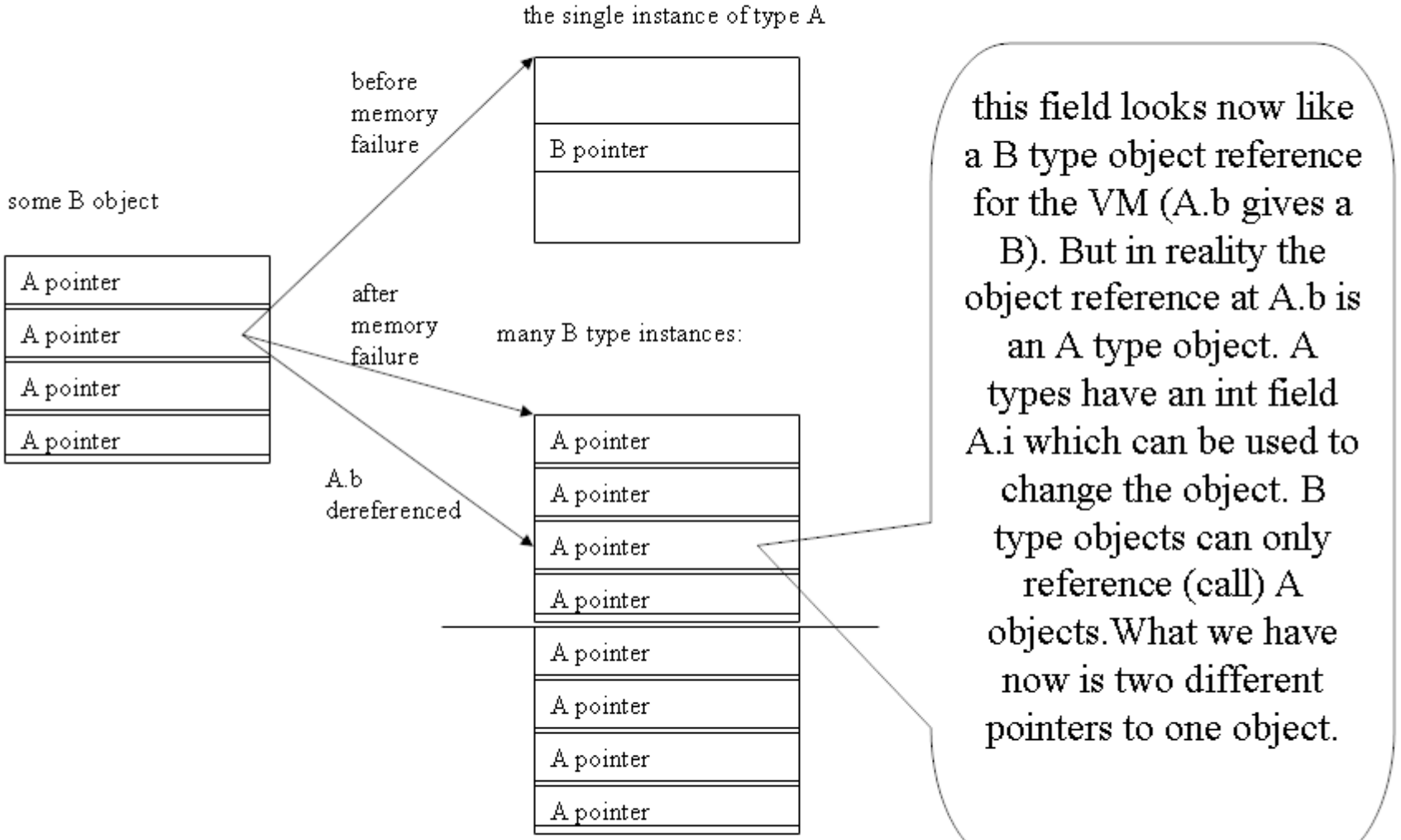
Object : 32 bytes, field : 4 bytes, header : 4 bytes. (from Govindavajhala…)

memory layout for VM type safety attack

- Inviting to attack type safety.
  - Segment size : Data >> text
- Attack applet allocates a lot of memory like this
- Applet waits for a memory error
- Random pointer p.b dereference will fetch from an A field.
- Type safety violated
  - Pointer of type B points to an A object. But it can also be casted again to an A object (runtime).



B and A pointer to same object



Exploit Code

```
class A {
        A a1;
        A a2;
        B b;
        int i;
        A a5;
        A a6;
        A a7;
}
```

```
// p == q
A p;
B q;
int offset = 4 * 4;
void write(int address, int value) {
p.i = address - offset ;
q.a4.i = value ;
)
```

```
class B {
        A a1;
        A a2;
        A a3;
        A a4;
        A a5;
        A a6;
        A a7;
}
```

p.i sets the address of an object of type A. When this A reference is used with a4.i the real access would be i – base of A. So this offset is subtracted first. (from Govindavajhala…)

Resources (1)

- Computer Architecture, John L. Hennessy, David A. Patterson. What you always wanted to know... from the fathers of RISC cpu‘s.
- Gary McGraw, John Viega on buffer overflows: (see also www.ibm.com/developerworks and search for buffer overflow)

  - learning the basics of buffer overflows: http://www-109.ibm.com/cgi-bin/click.pl?url=http://www-106.ibm.com/developerworks/library/overflows/index.html&qry=buffer%20overflow

  - preventing buffer overflows http://www-109.ibm.com/cgi-bin/click.pl?url=http://www-106.ibm.com/developerworks/library/buffer-defend.html&qry=buffer%20overflow (Shows which c-funtions are dangerous and how to substitute them)

  - brass tacks and smash attacks: http://www-109.ibm.com/cgi-bin/click.pl?url=http://www-106.ibm.com/developerworks/library/smash.html&qry=buffer%20overflow (this shows EXACTLY how it works)

  - An anatomy of attack code: http://www-109.ibm.com/cgi-bin/click.pl?url=http://www-106.ibm.com/developerworks/library/attack.html&qry=buffer%20overflow

Resources (2)

- A comparative Analysis of Methods of Defense against Buffer Overflow Attacks, Istvan Simon, www.mcs.csuhayward.edu/~simon/security/boflo.html
- Libsafe: Protecting Critical Elements of Stacks, Arash Baratloo et.al, www.bell-labs.com/org/11356/libsafe.html
- Jochen Liedtke, Uni Karlsruhe, System Architecture and Solution 1 (System Architecture Group). Nice stack layouts.
- Sudhakar Govindavajhala and Andrew W. Appel, Princeton University July 2003: Using Memory Errors to attack a Virtual Machine. Nice article on how to use (create) physical memory damages and how one bit is enough to break Virtual Machine type safety – and as a result of this memory safety. Arbitrary memory locations can be written once 2 different object types point to one memory location. www.cs.princeton.edu/~sudhakar/talks/memerr.ppt and http://www.cs.princeton.edu/~sudhakar/papers/memerr.pdf

Resources (3)

- Peter Silberman, Richard Johnson, A comparison of buffer overflow prevention implementations and weaknesses. Excellent paper from www.idefense.com on kernel and compiler technology to prevent the execution of injected code. On a simpler, OS specific level see also:
- Oliver Lavery, Win32 message vulnerability redux – shatter attacks remain a threat. Very nice paper from www.idefense.com on how to exploit window messages on windows OS and why privileged services running a GUI are dangerous (anyway, besides implementation flaws)
- Solar-Designer explanation of return-to-libc mechanism. www.groar.org/expl/intermediate/ret-libc.txt
- Mozilla.org vulnerability list. Allows analysis of kind and frequency of certain software problems causing security vulnerabilities. http://www.mozilla.org/projects/security/known-vulnerabilities.html
- Alexander Sotirov, Mark Dowd, „Bypassing Browser Memory Protections – setting back browser security by 10 years“, Black Hat Conference August 2008