Why are we here?

Why are we here?

https://news.ycombinator.com/item?id=7836283

Y **Hacker News** new | comments | show | ask | jobs | submit

▲ wtracy 531 days ago | parent

It is unethical to continue writing code in non-memory-safe C or C-based languages, for any purpose. Period.

I'm looking forward to seeing your new operating system and managed runtime written entirely using garbage-collected languages!

Why are we here?

https://news.ycombinator.com/item?id=7836283

Hacker News new | comments | show | ask | jobs | submit

wtracy 531 days ago | parent

It is unethical to continue writing code in non-memory-safe C or C-based languages, for any purpose. Period.

I'm looking forward to seeing your new operating system and managed runtime written entirely using garbage-collected languages!

http://blog.regehr.org/archives/715#comment-4242

Greg A. Woods | May 22, 2012 at 11:13 pm | Permalink

I don't expect C to ever become memory safe unless it becomes something I would not feel comfortable calling "C", and I would really hate it if people did start pretending it is "C".

LLVM

in the context of

Software Safety

How are our programs attacked?

Which LLVM tools enhance software safety?

Inga Rüb (inga.roksana.rueb@gmail.com)

LLVM

in the context of

Software Safety

How are our programs attacked?

Which LLVM tools enhance software safety?

Inga Rüb (inga.roksana.rueb@gmail.com)

LLVM

in the context of

Software Safety

How are our programs attacked?

Which LLVM tools enhance software safety?

Inga Rüb (inga.roksana.rueb@gmail.com)

Let’s talk about:

- Bases of memory management for a single process
- Exemplary exploits: how-to
- LLVM tools that prevents some of software attacks

Let’s talk about:

- **Bases of memory management for a single process**
- Exemplary exploits: how-to
- LLVM tools that prevents some of software attacks

Let’s talk about:

- **Bases of memory management for a single process**
- Exemplary exploits: how-to
- LLVM tools that prevents some of software attacks

Presented information refers to Linux 32-bit distributions.

Let’s talk about:

- **Bases of memory management for a single process**
- Exemplary exploits: how-to
- LLVM tools that prevents some of software attacks

because of personal preferences

Presented information refers to Linux 32-bit distributions.

because of the level of complexity

Bases of memory management

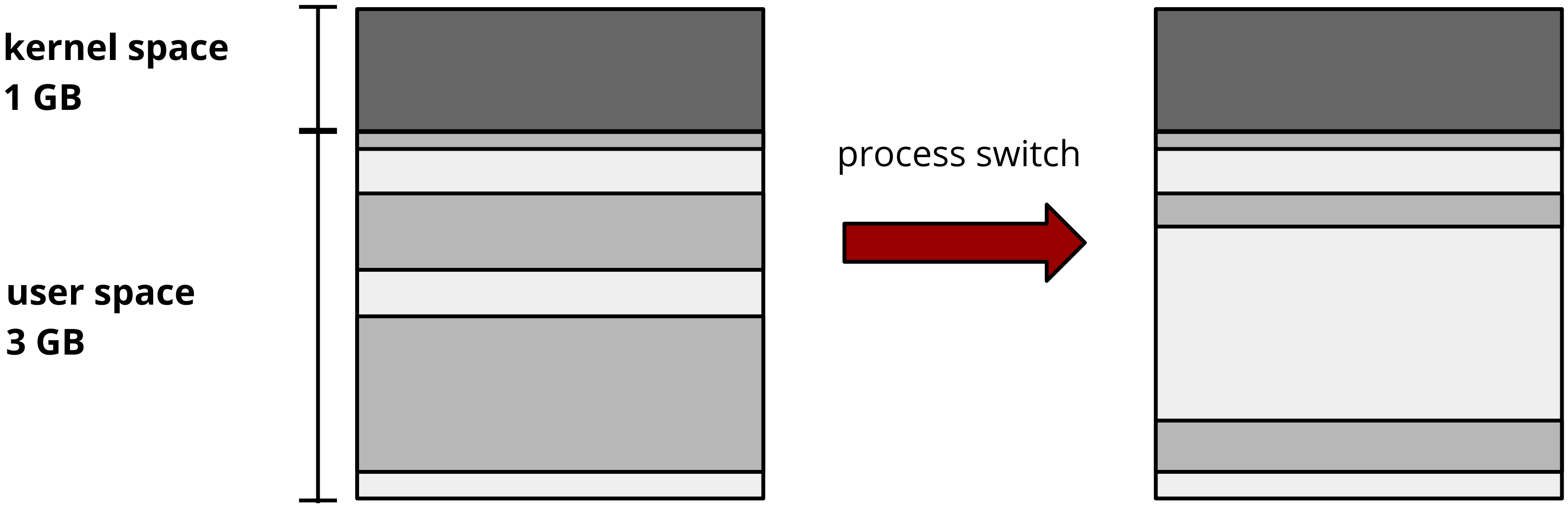
Structure of process memory

Calling convention

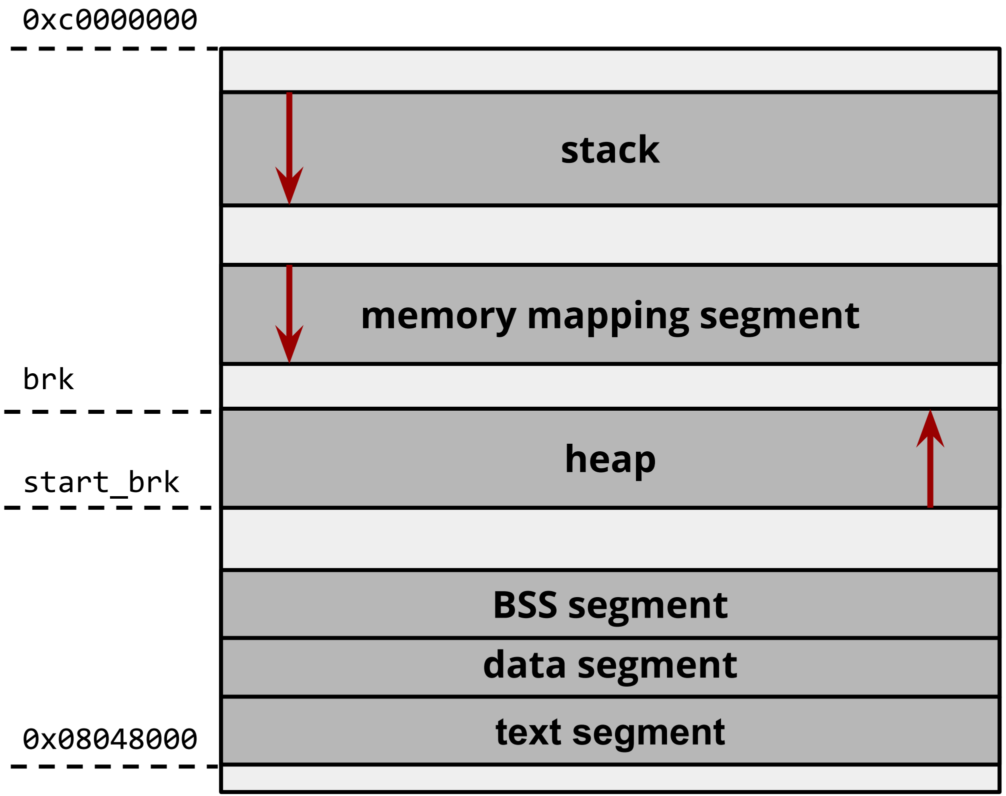
Virtual memory

At a time there is 4GB of virtual memory reserved.

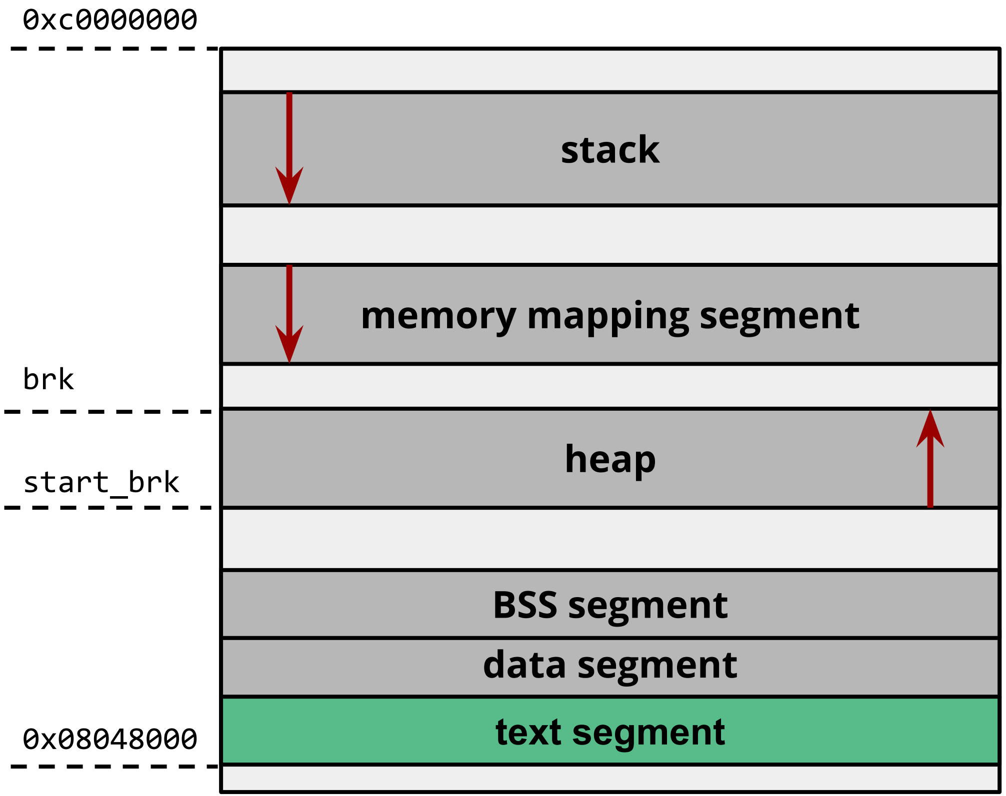
Virtual addresses are mapped to physical memory.



User space


0xc0000000
stack
memory mapping segment
brk
heap
start_brk
BSS segment
data segment
text segment
0x08048000

User space



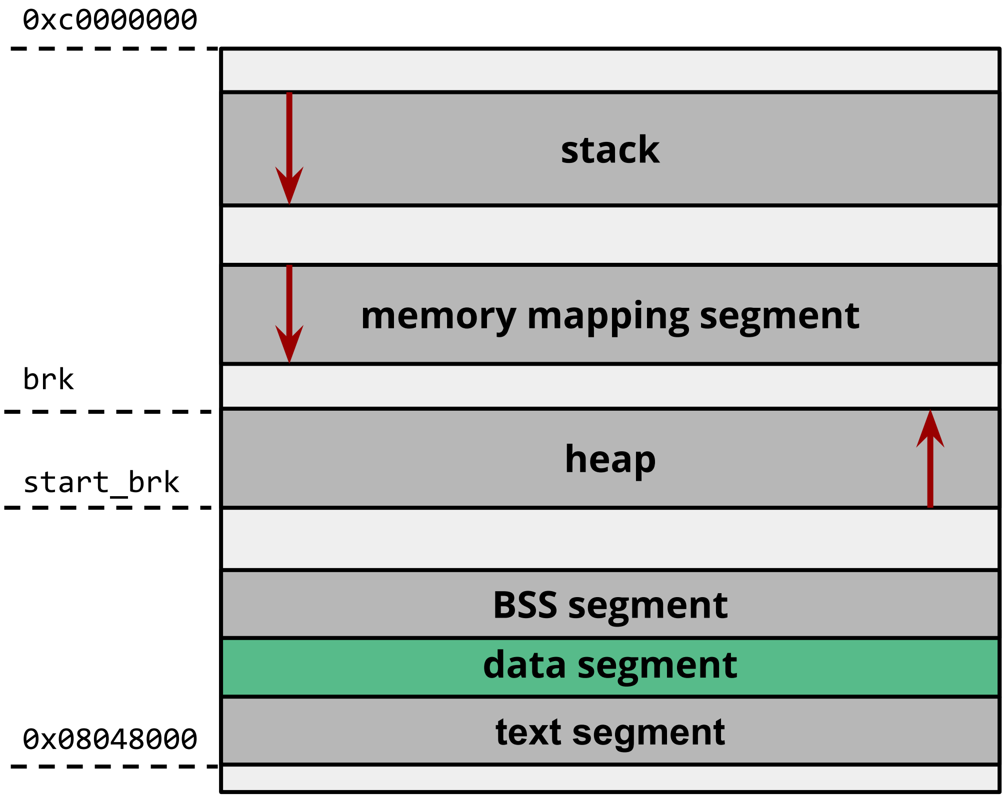
text segment (code segment)

read-only

fixed size

corresponds to a part of an object file and contains executable instructions

User space



data segment

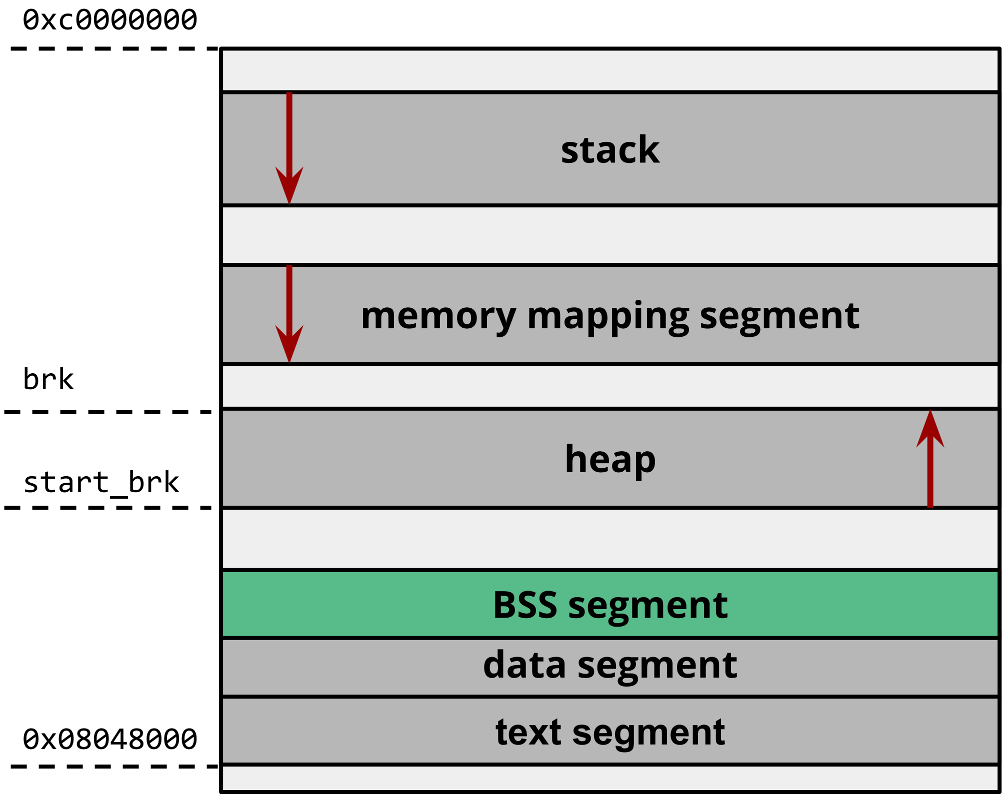
read-write

fixed size

maps a file - private memory mapping

contains any global or static variables which have a predefined value and can be modified

User space



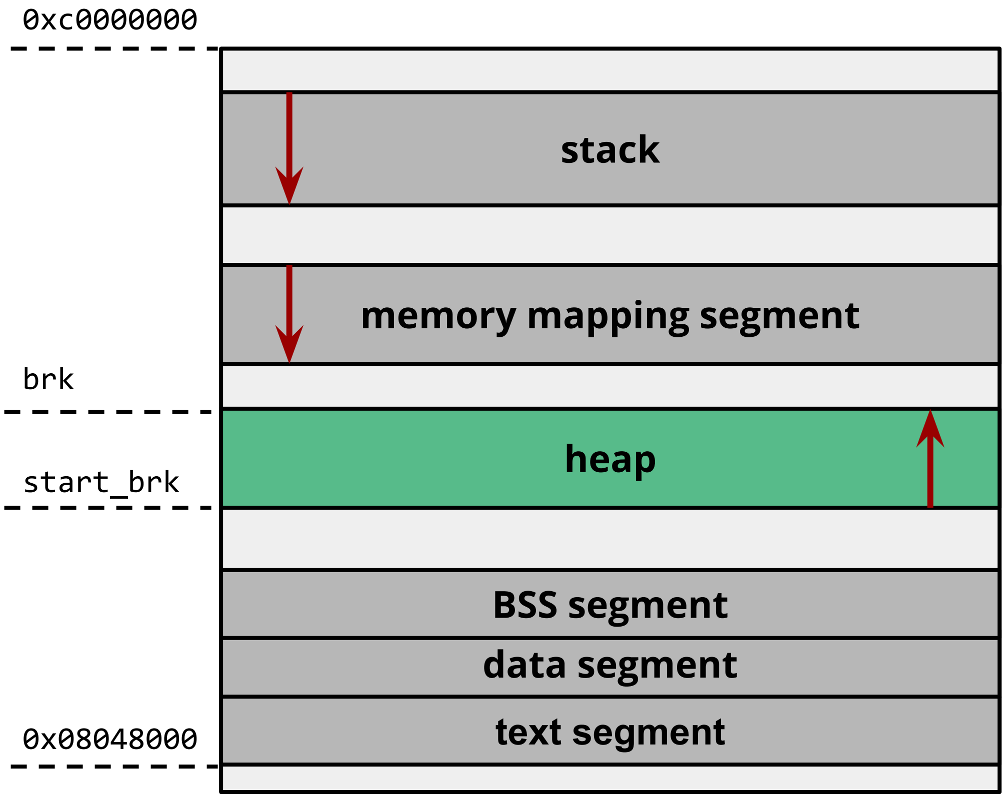
BSS segment

read-write

fixed size

memory initialized with zeroes that represent uninitialized static variables

User space



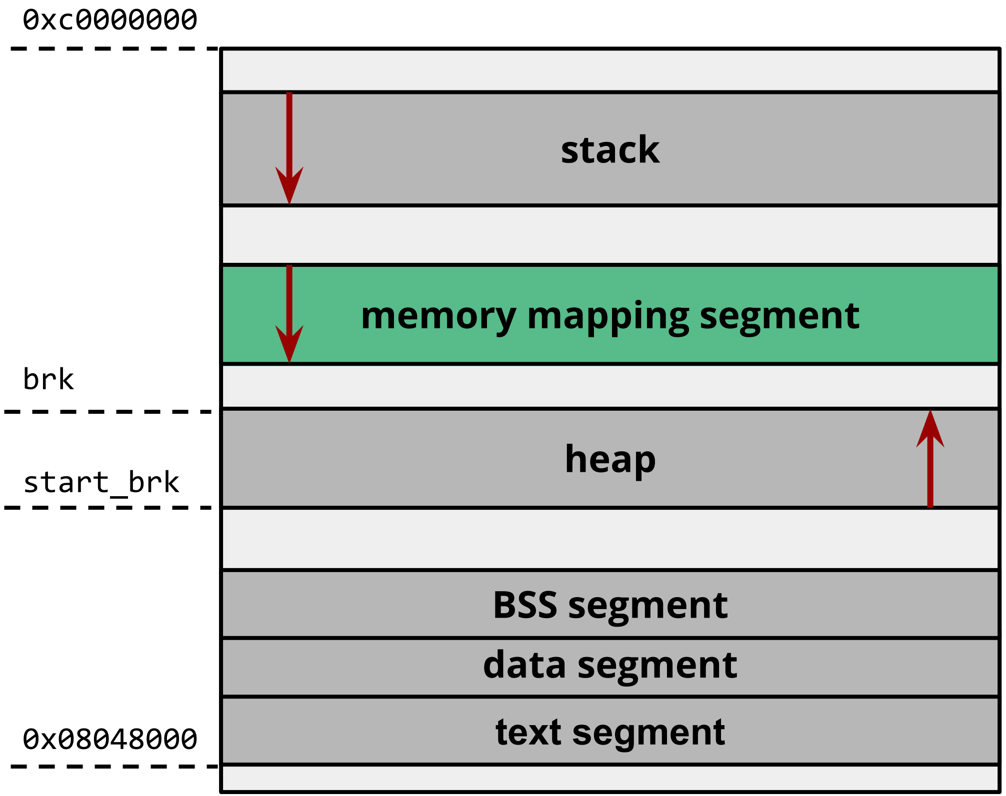
heap

read-write

dynamic size

contains the dynamically allocated memory

User space



memory management segment

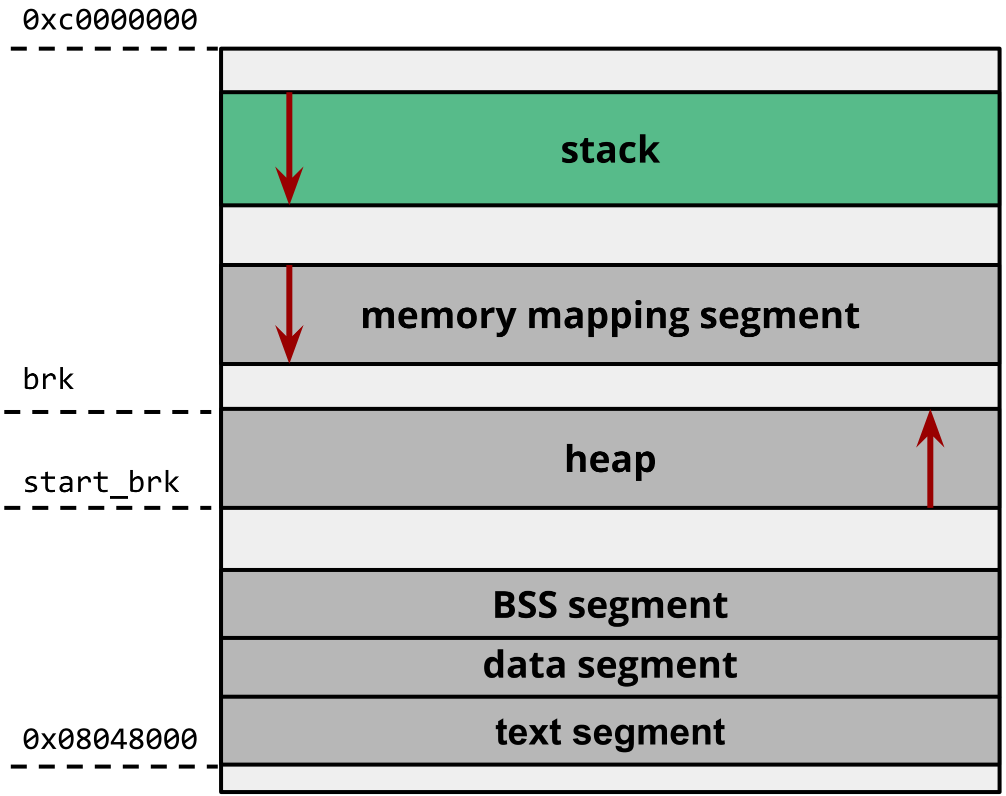
read-write

dynamic size

a direct byte-for-byte correlation with some portion of a file or file-like resource

`mmap()` syscall

User space



stack

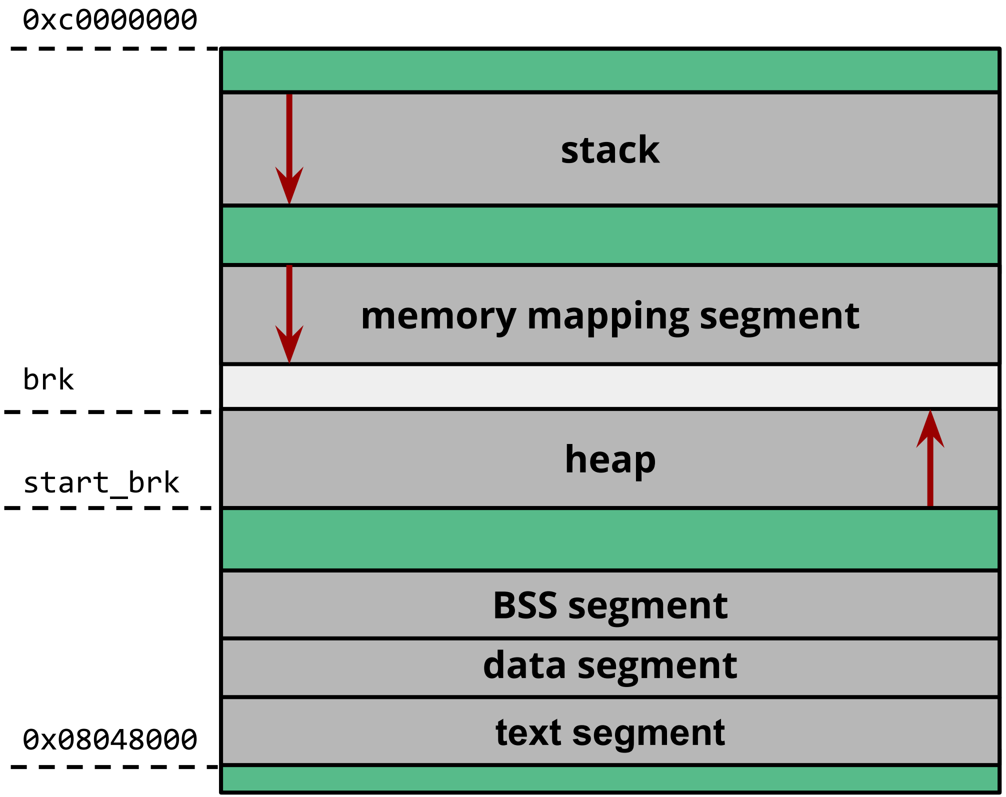
read-write

dynamic size

its size can be extended but is there is a limit (RLIMIT_STACK)

stores local variables and function parameters

User space



offsets

dynamic size (initially: random)

any access triggers a segfault

present because of safety reasons

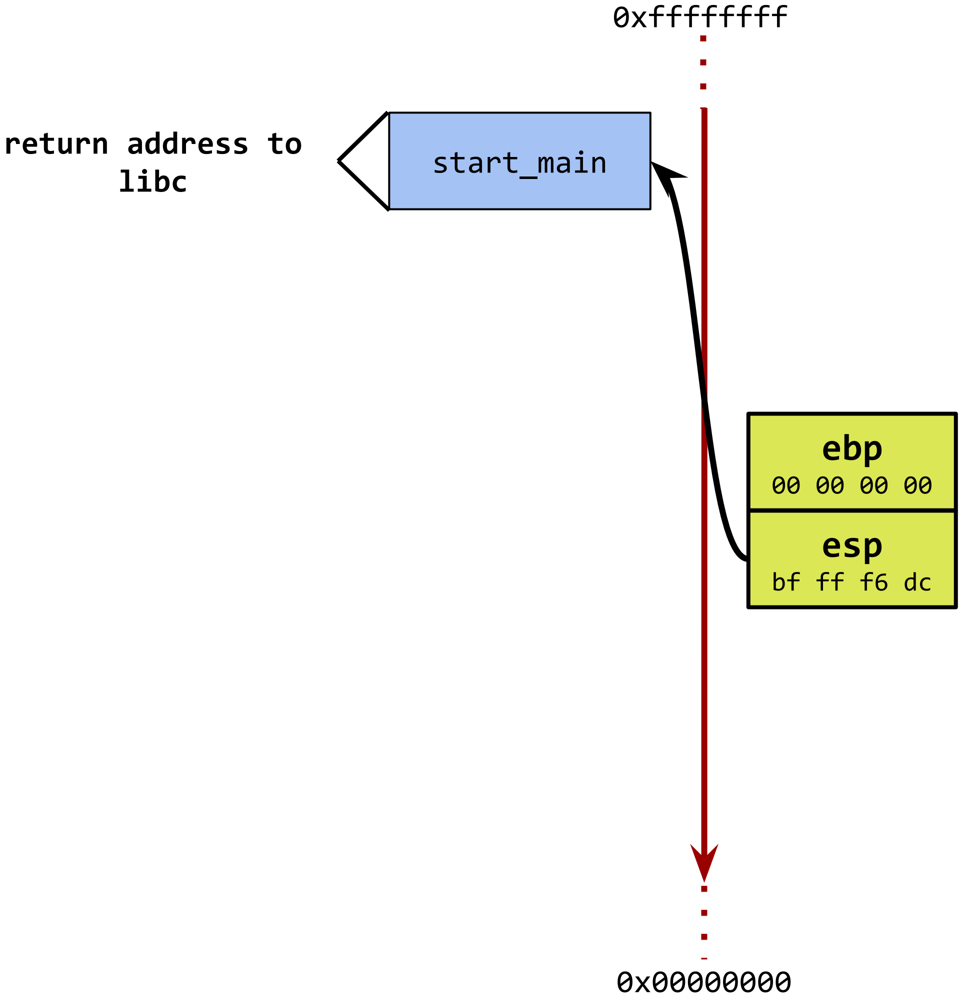
Bases of memory management

Structure of process memory

Calling convention

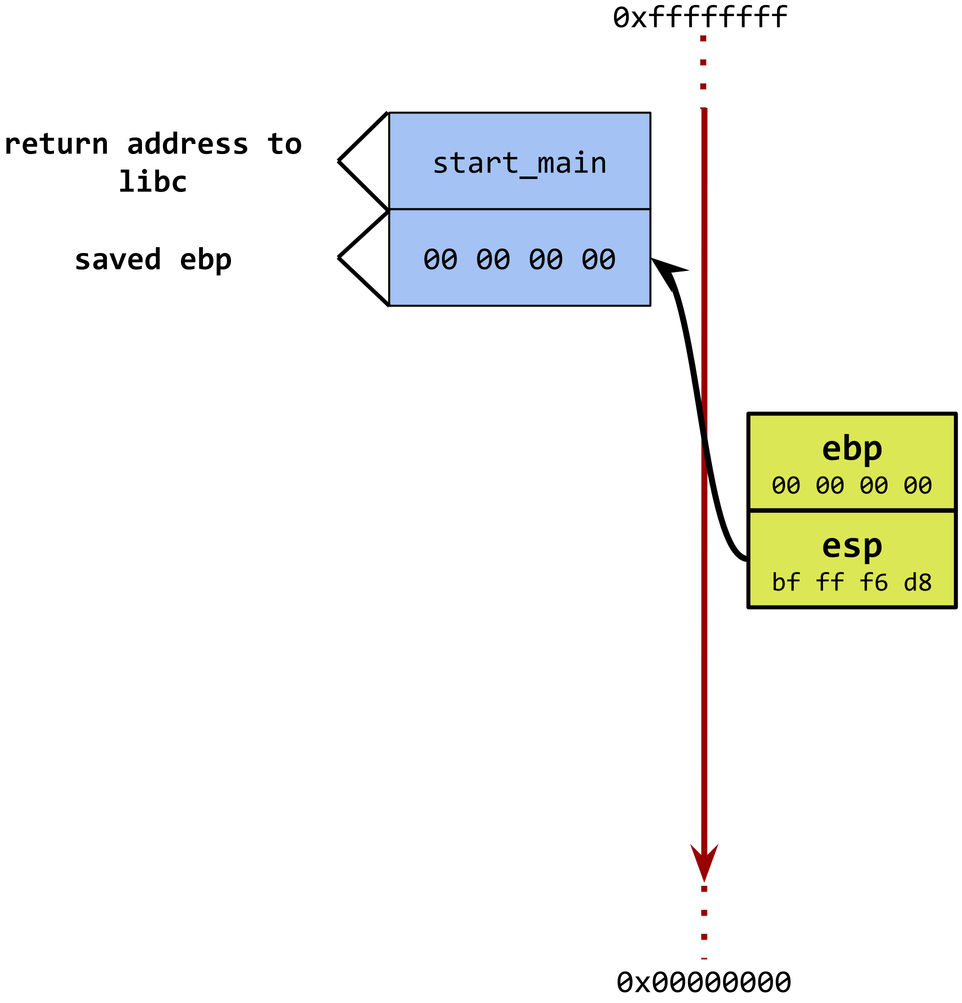
Stack frame - prolog

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



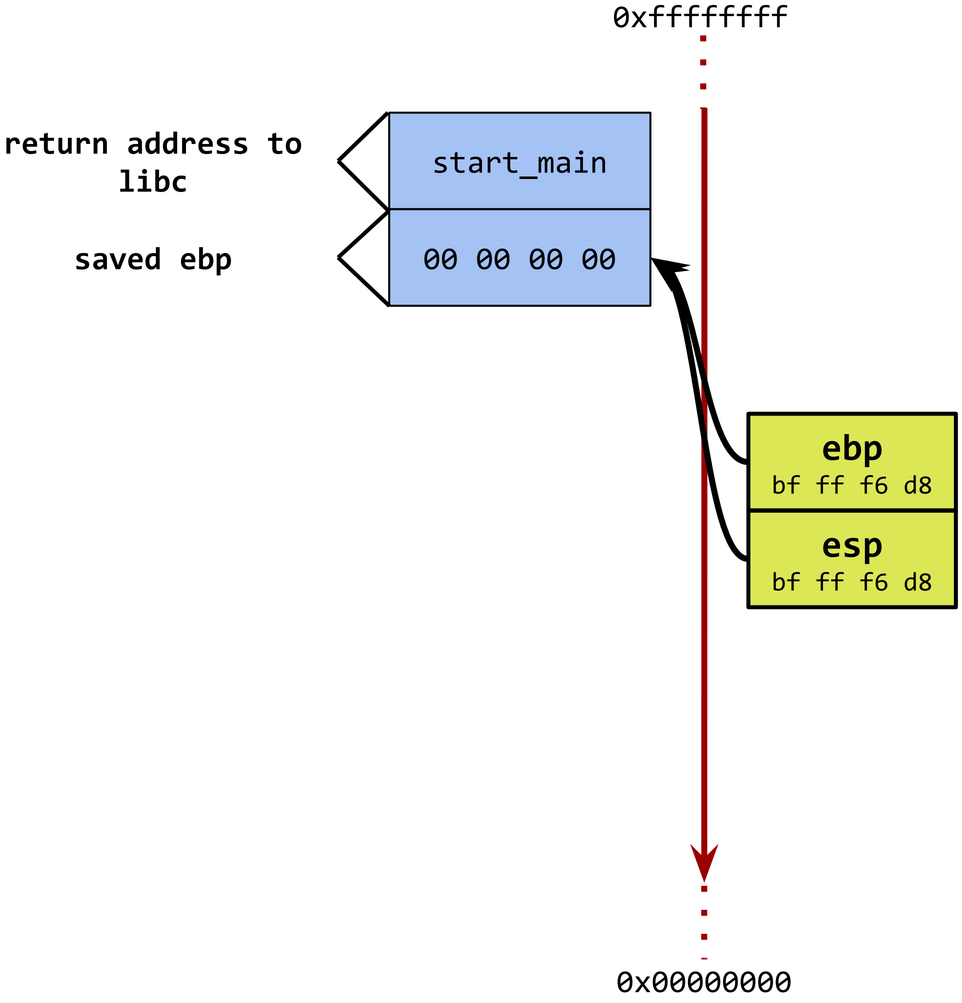
Stack frame - prolog

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



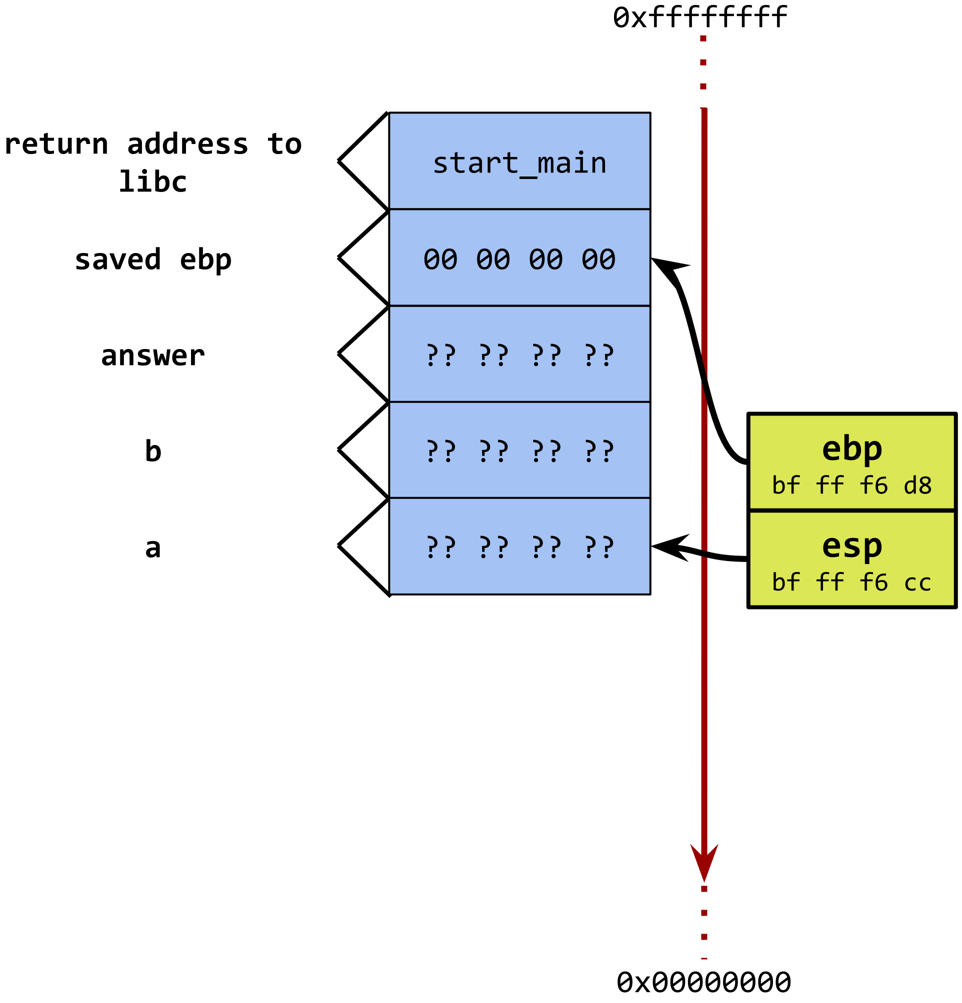
Stack frame - prolog

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



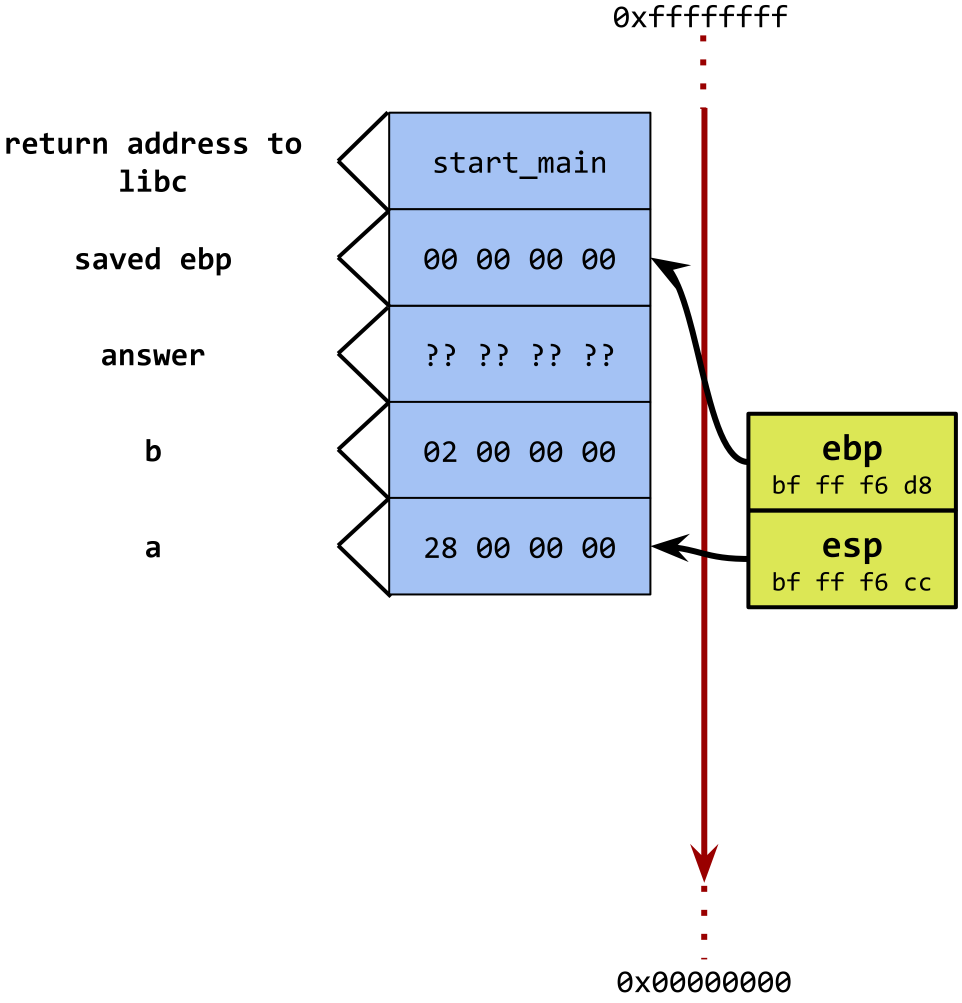
Stack frame - locals

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



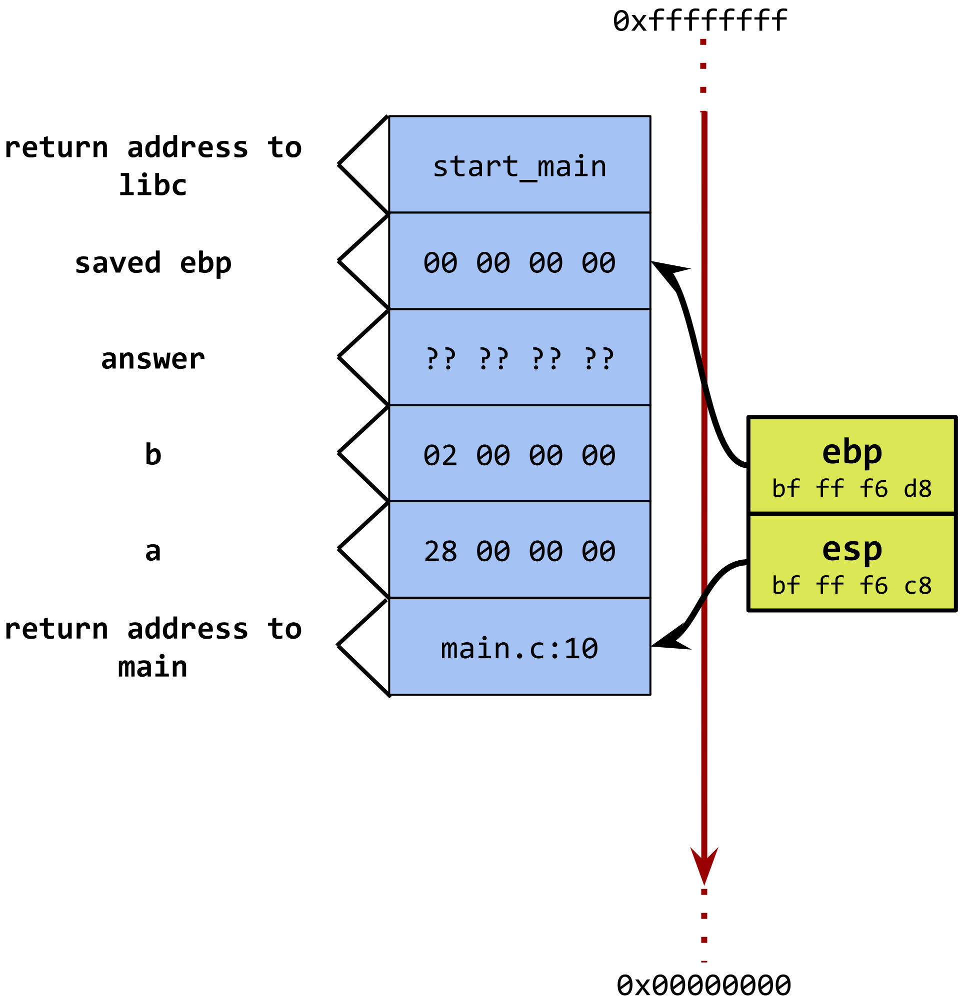
Stack frame - locals

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



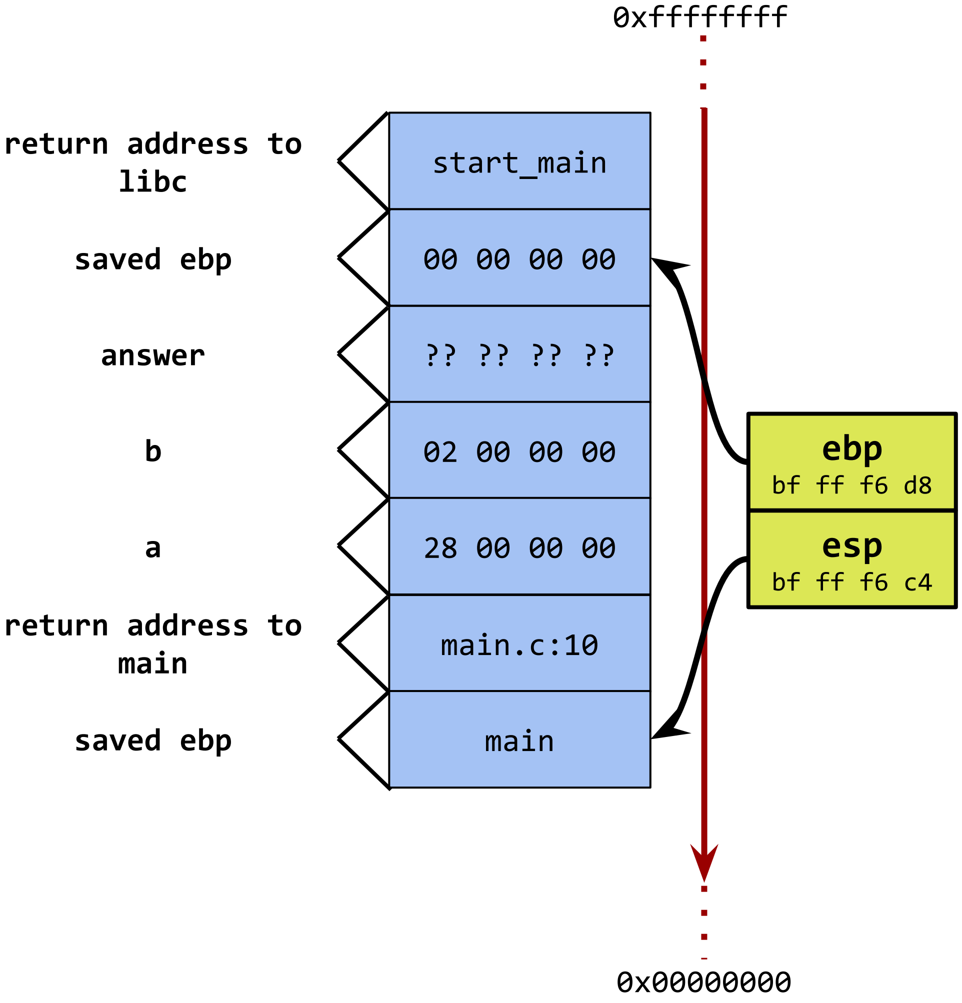
Stack frame - call

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



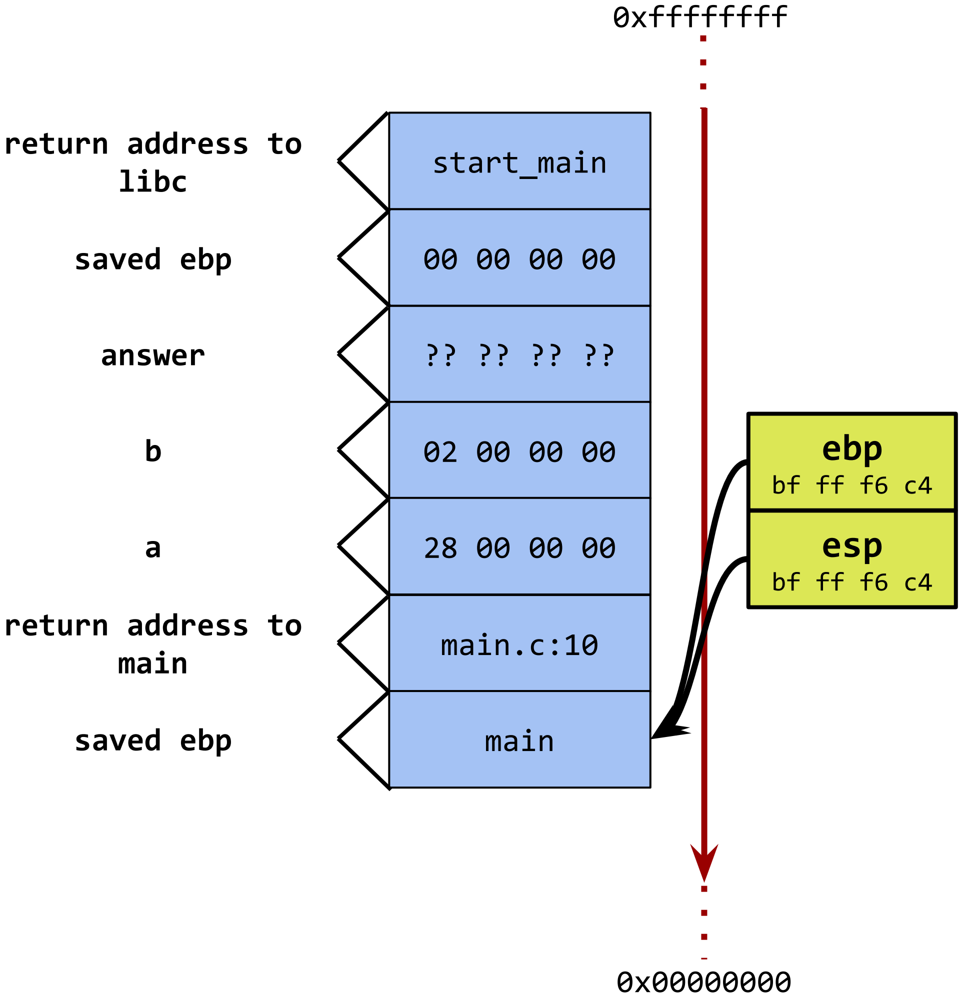
Stack frame - callee

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



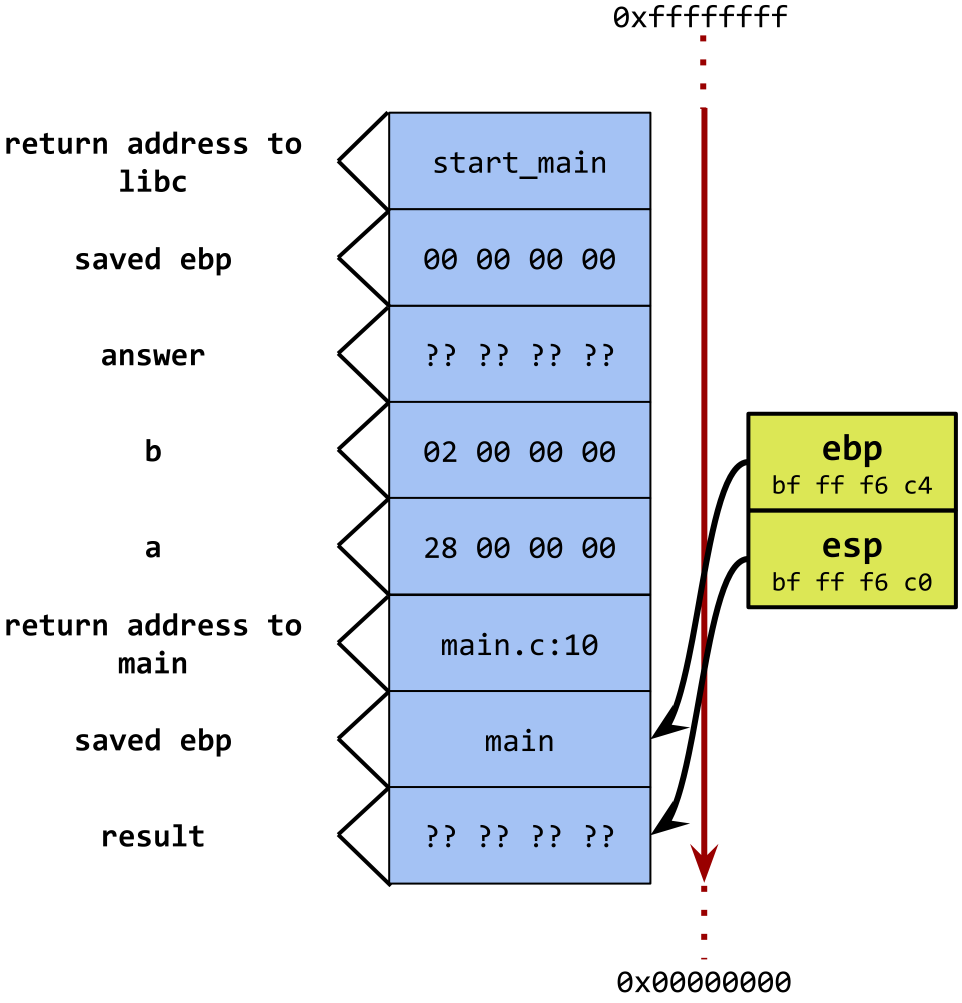
Stack frame - callee

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



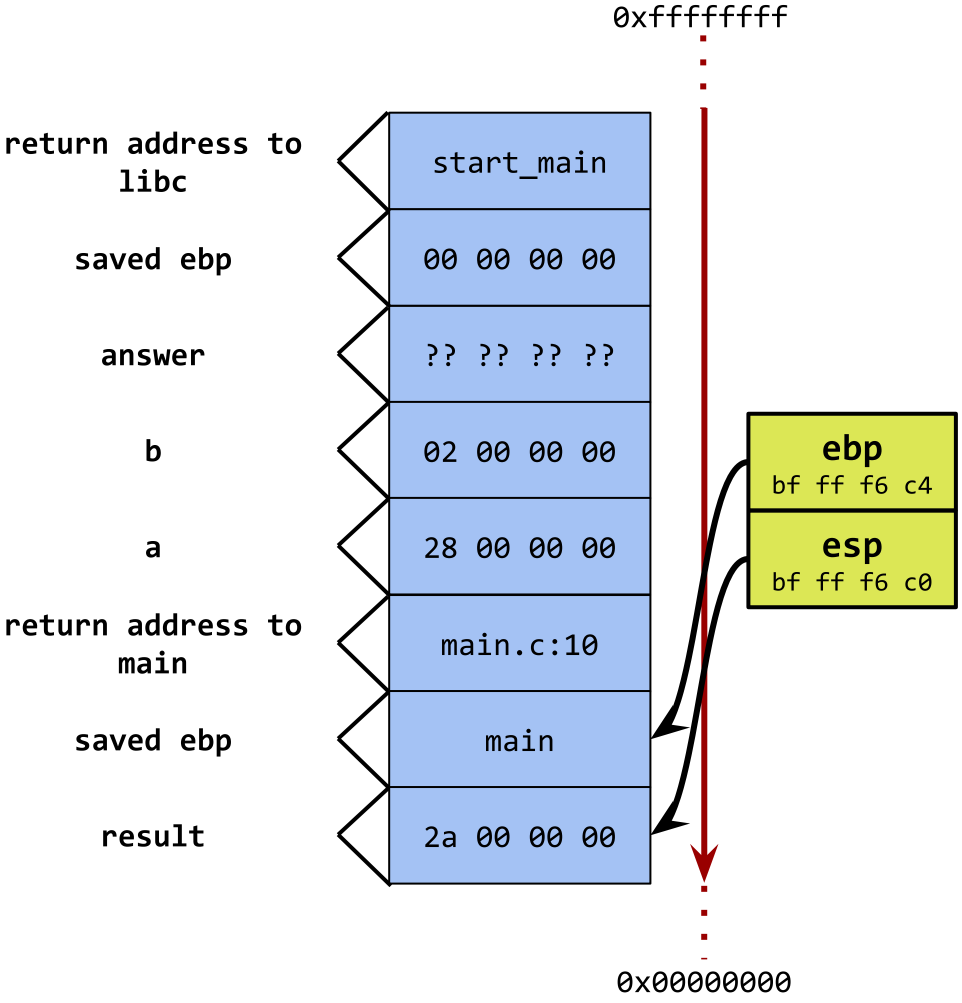
Stack frame - callee

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



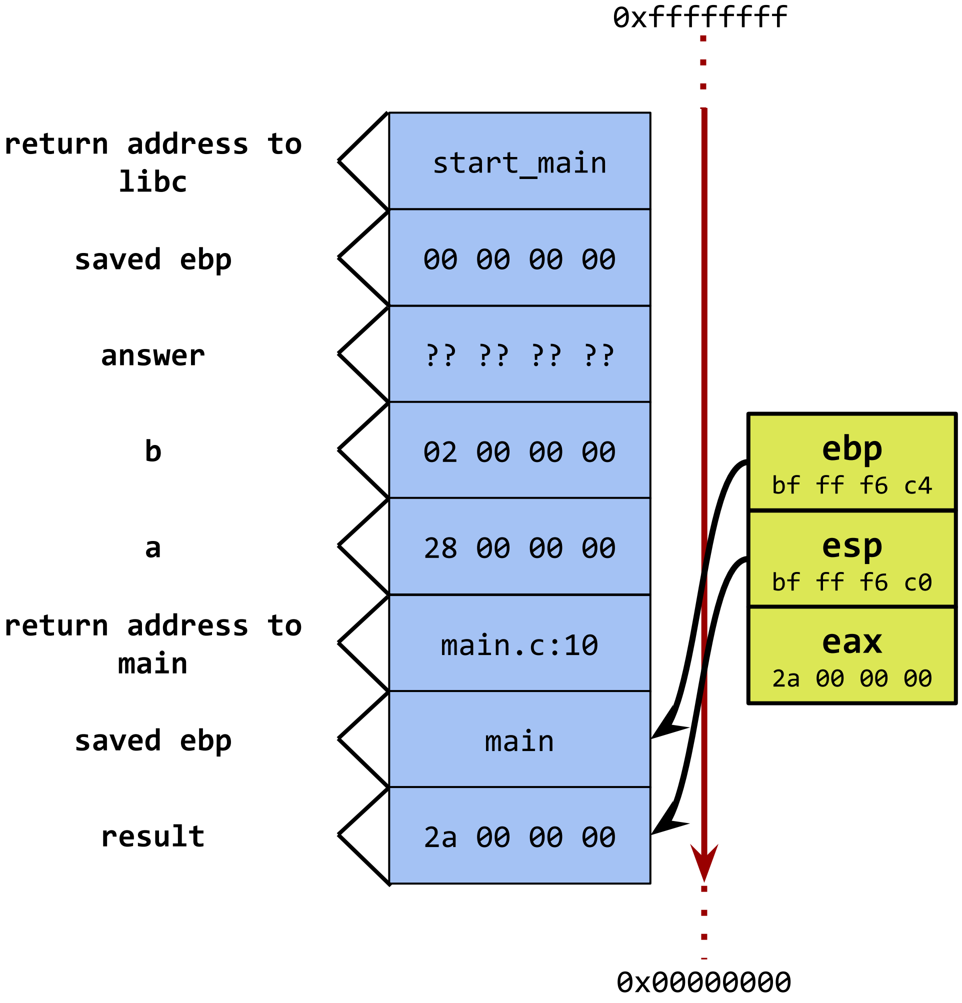
Stack frame - callee

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



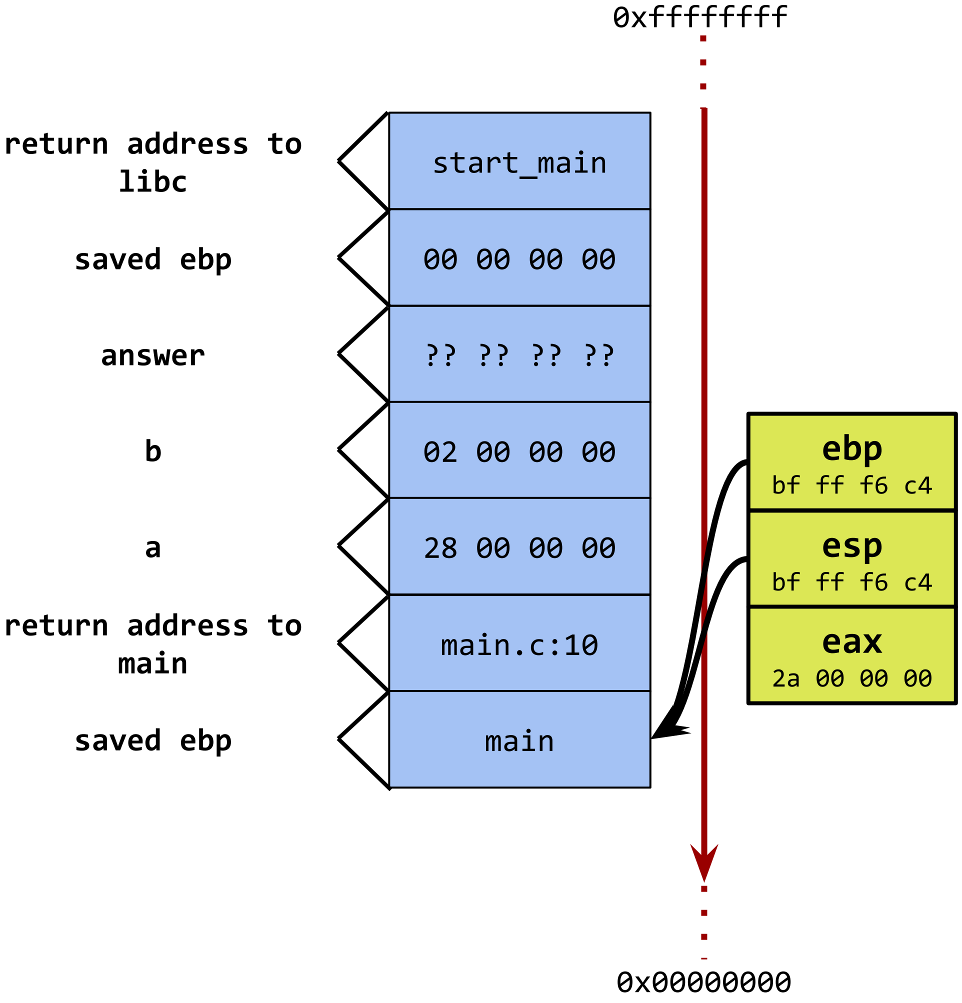
Stack frame - callee

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



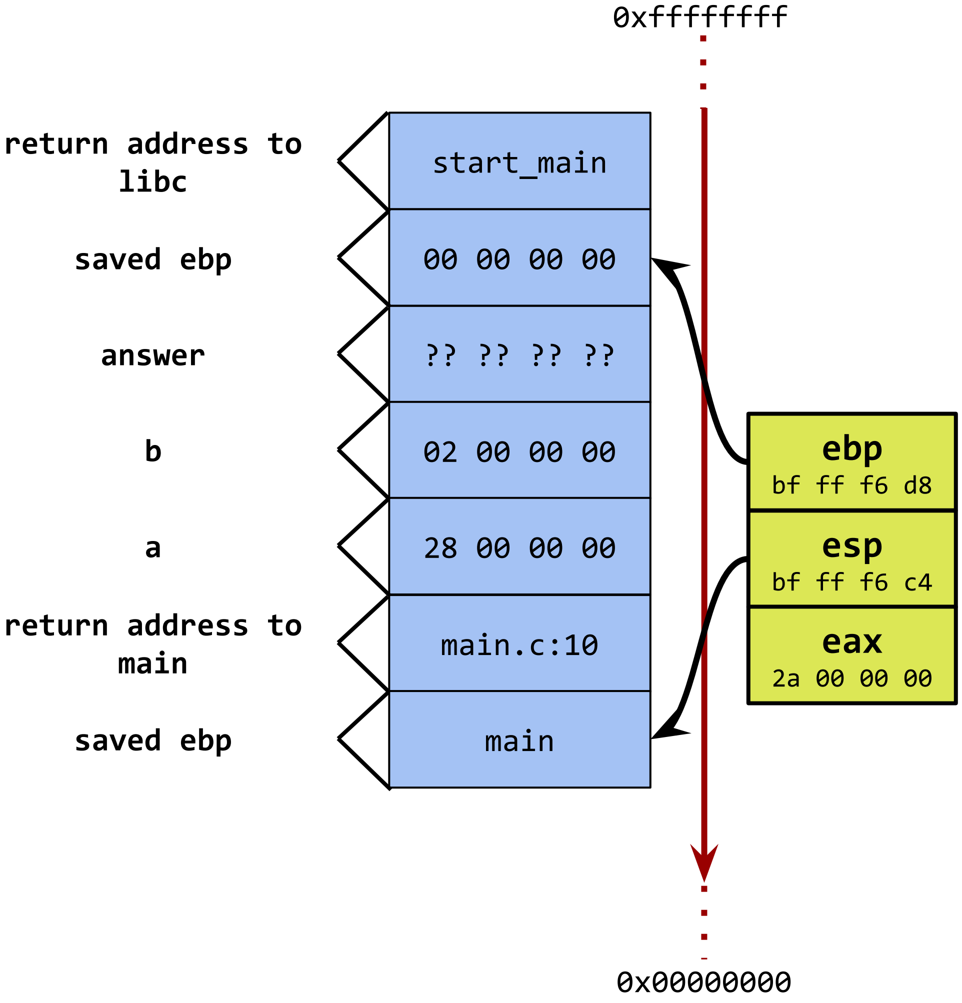
Stack frame - epilogue

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



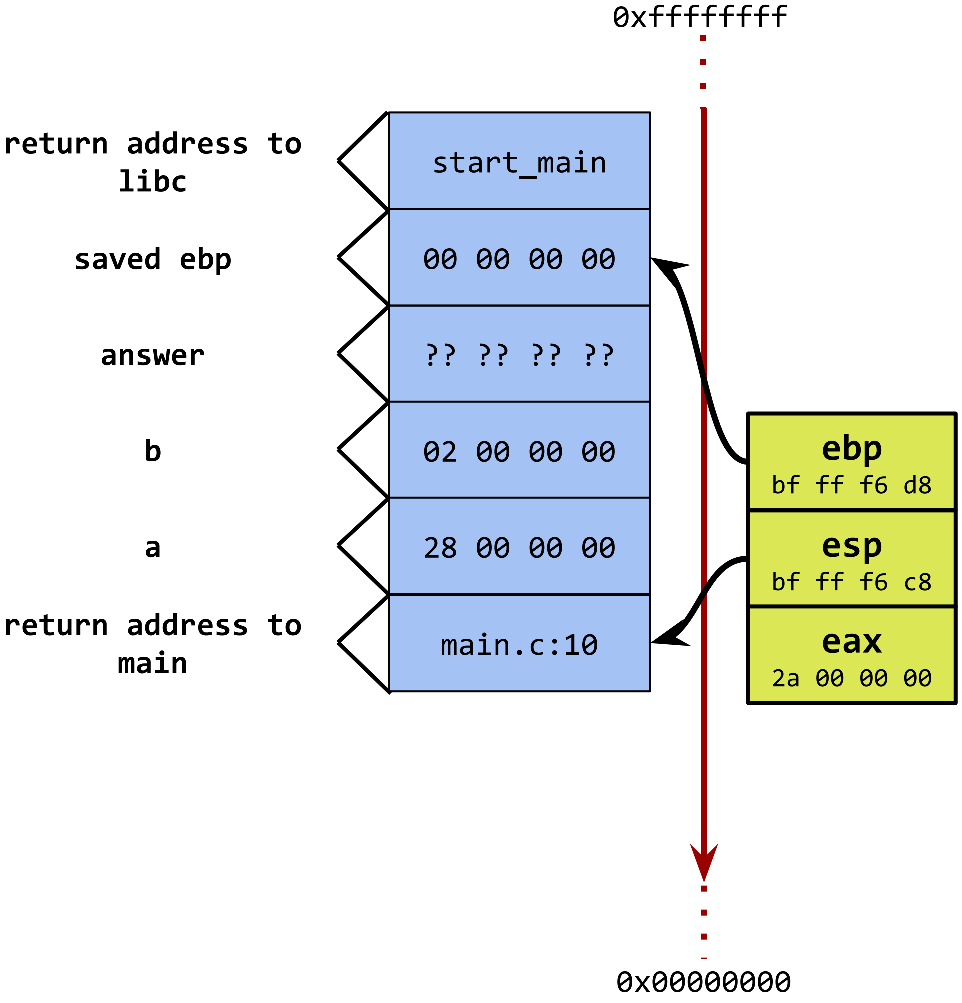
Stack frame - epilogue

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



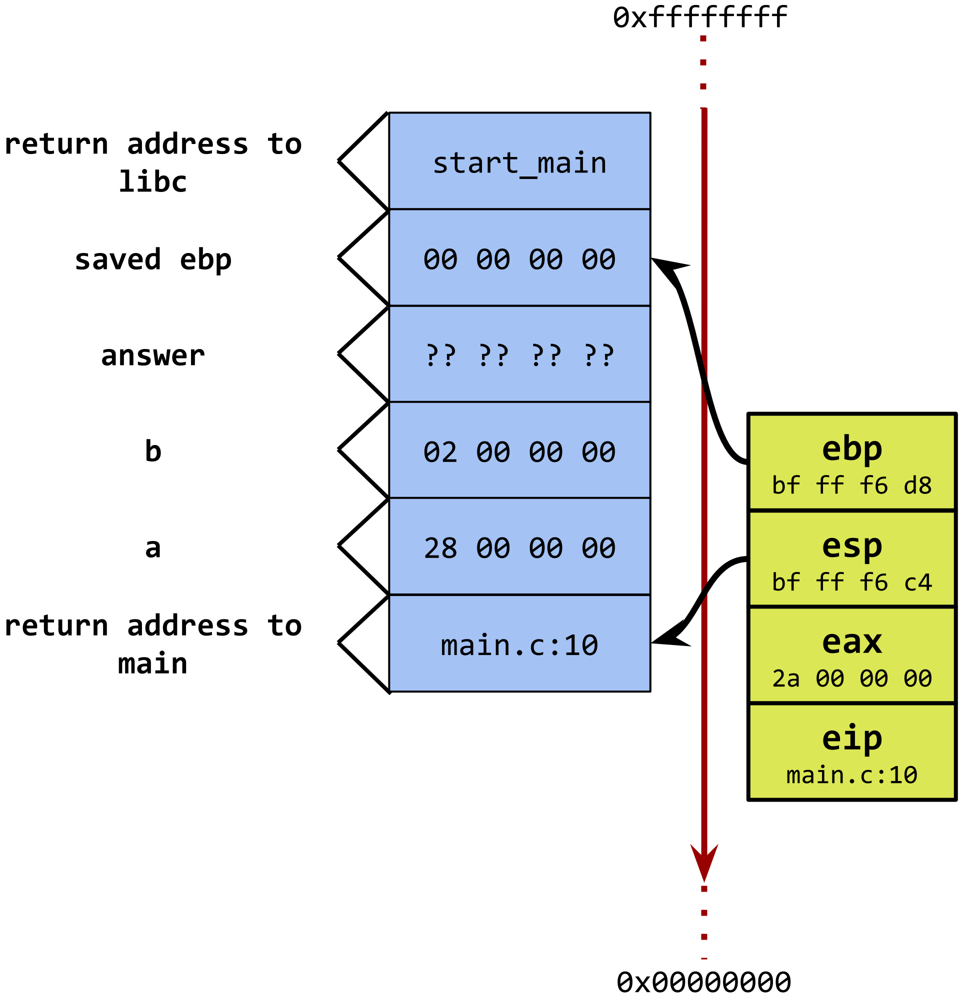
Stack frame - epilogue

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



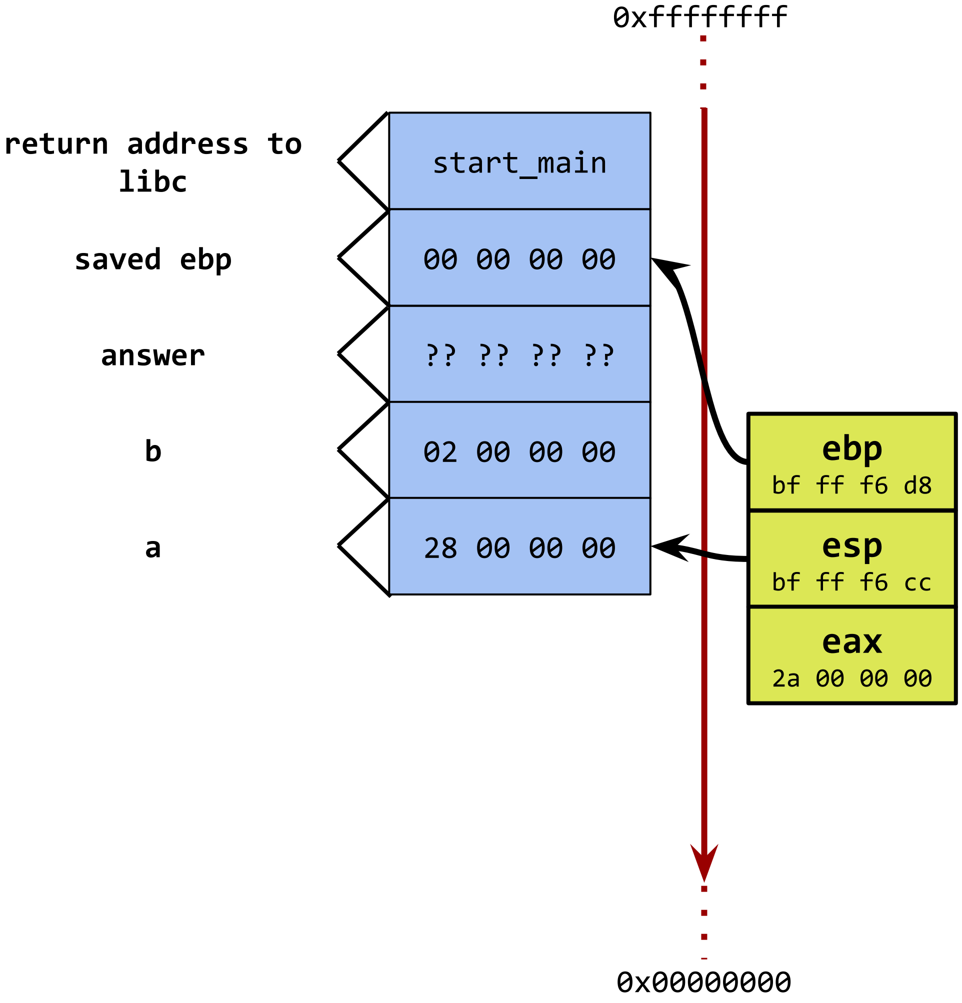
Stack frame - epilogue

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



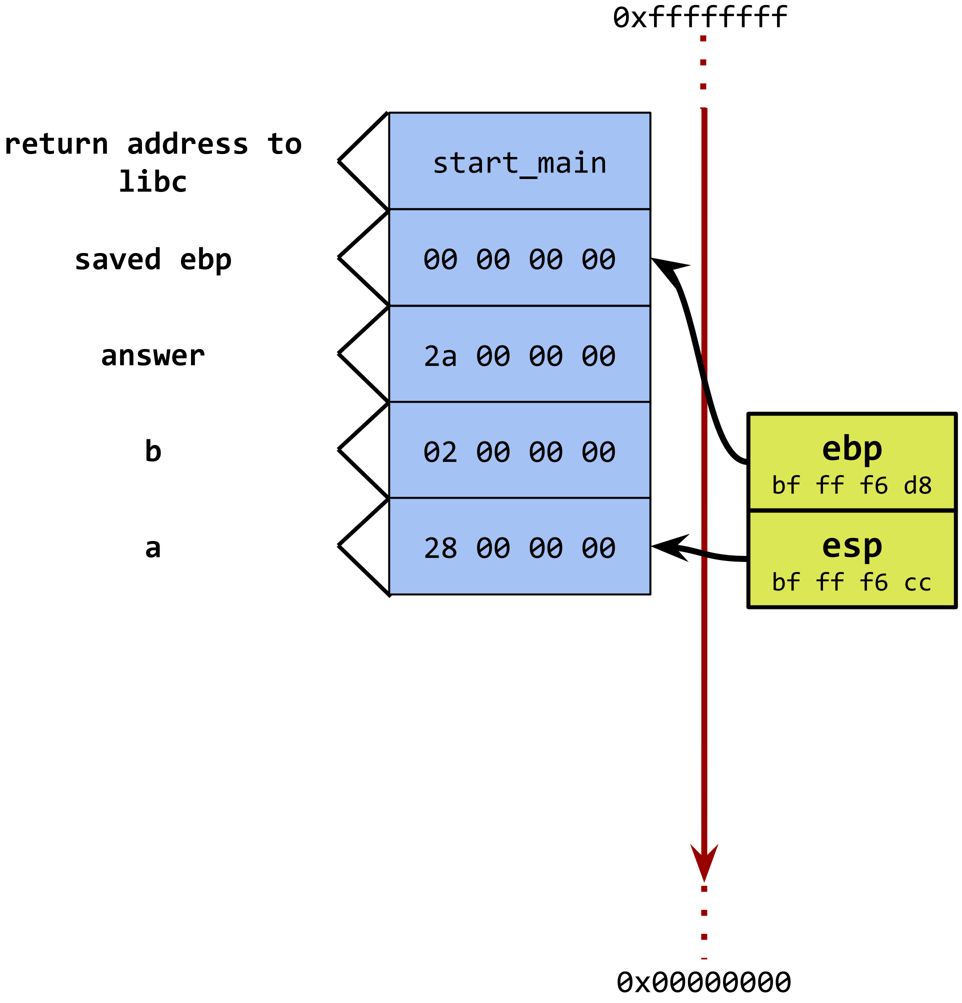
Stack frame - epilogue

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



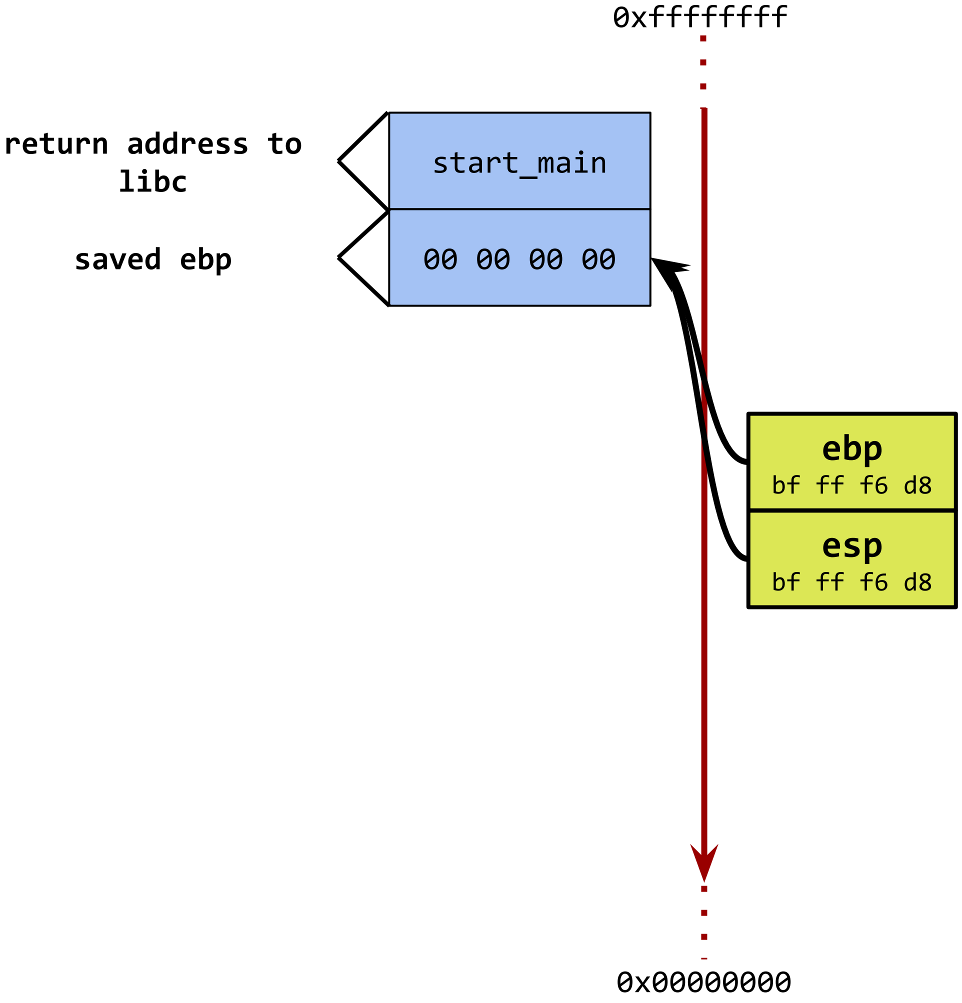
Stack frame - call

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



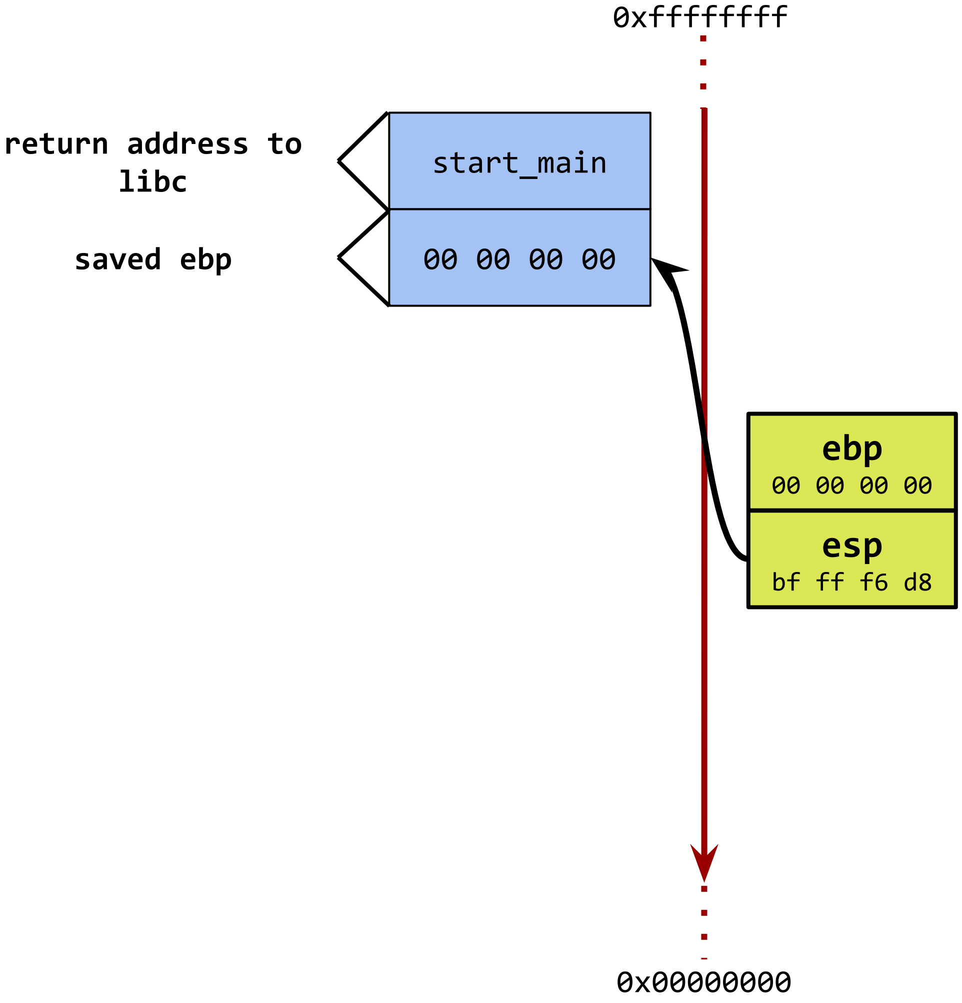
Stack frame - epilogue

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



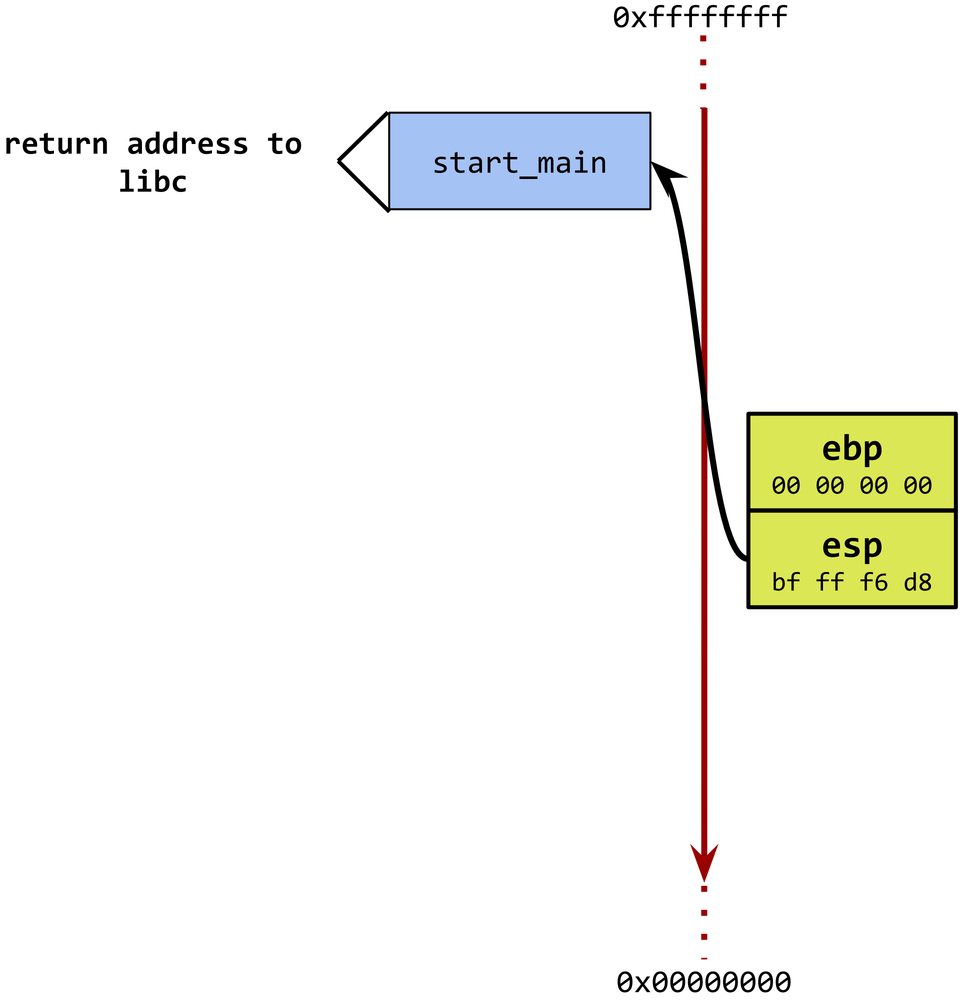
Stack frame - epilogue

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



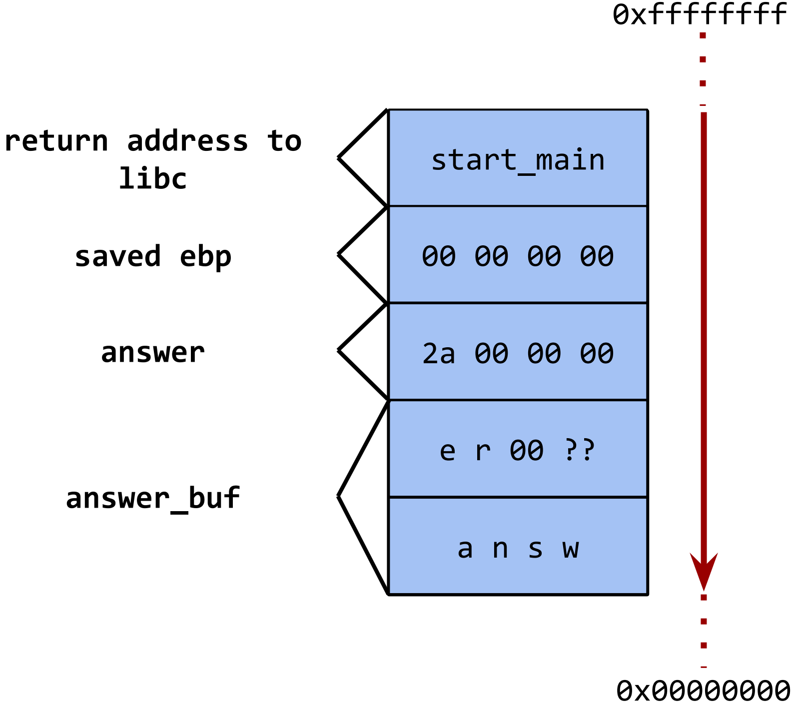
Stack frame - epilogue

```c
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main()
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



Buffers on the stack

```c
int main(int argc, char** argv)
{
  int answer = 42;
  char answer_buf[8] = "answer";
  return 0;
}
```



Buffers grow towards higher addresses.

References

Anatomy of a program in memory:

http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

User space memory access from the Linux kernel:

http://www.ibm.com/developerworks/library/l-kernel-memory-access/

Let’s talk about:

- Bases of memory management for a single process
- **Exemplary exploits: how-to**
- LLVM tools that prevents some of software attacks

Exemplary exploits: how-to

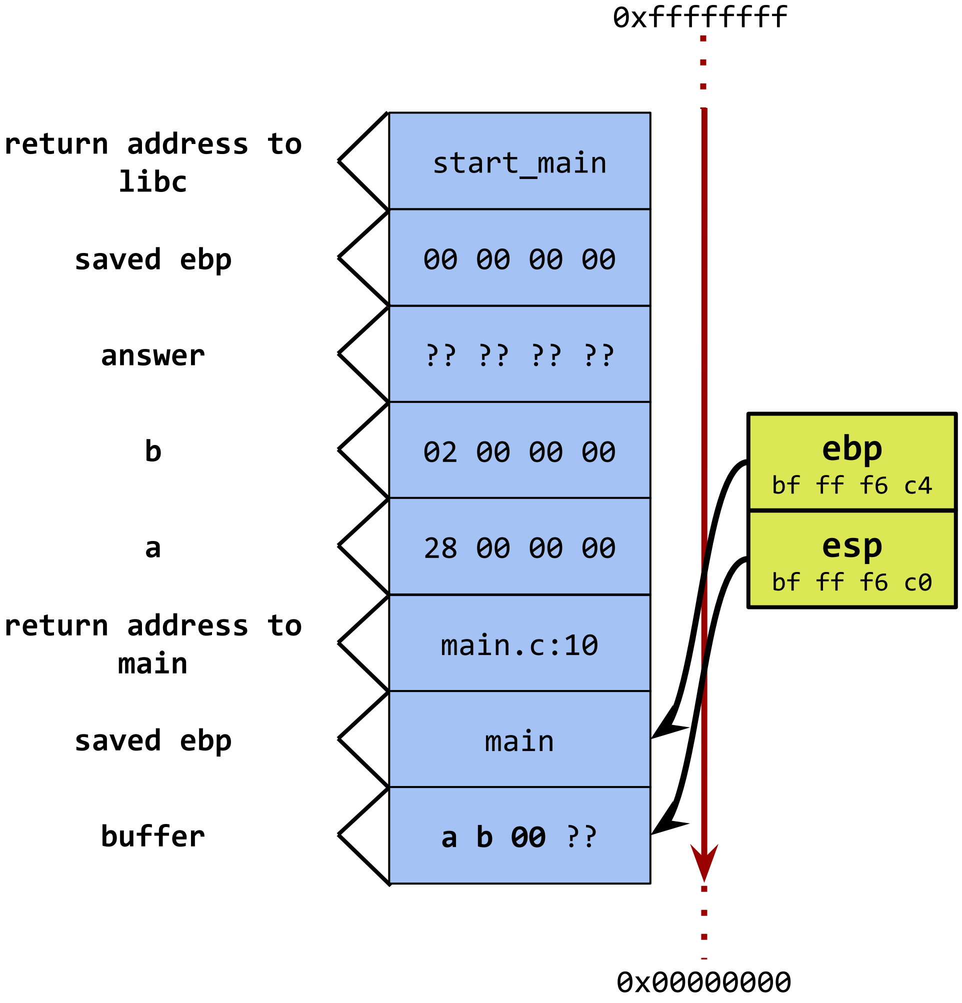
Use Return Oriented Programming

Defend against attacks

Define safety

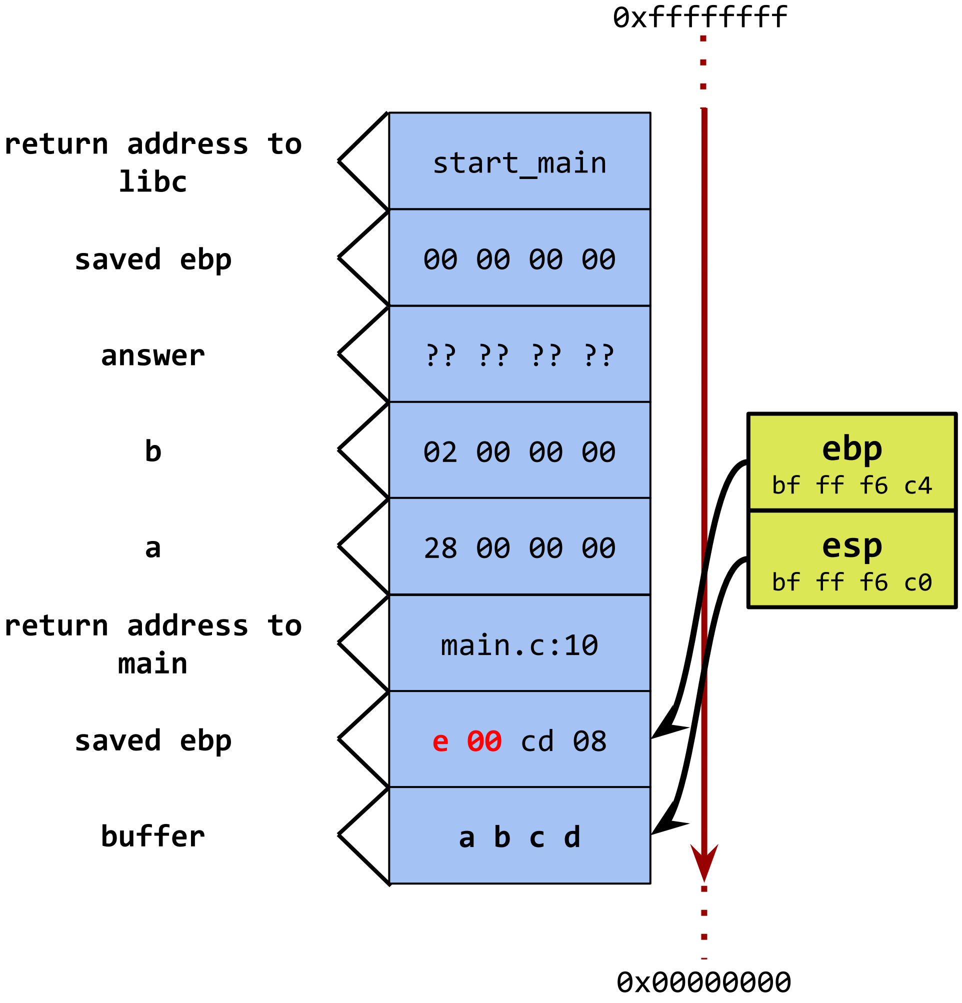
Buffer on the stack

```
int foo(int a, int b)
{
  char buffer[4] = “ab”;
  return 42;
}

int main(int argc, char** argv)
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



Buffer overflow

```
int foo(int a, int b)
{
  char buffer[4] = "abcde";
  return 42;
}

int main(int argc, char** argv)
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



Return oriented programming

Return oriented programming

an attacker uses his control over the stack

Return oriented programming

an attacker uses his control over the stack

right before the return from a function

Return oriented programming

an attacker uses his control over the stack

right before the return from a function

to direct code execution to some other location in the program

How critical are memory corruption bugs?

MITRE ranking [http://cwe.mitre.org/top25/]:

*The 2011 CWE/SANS Top 25 Most Dangerous Software Errors is a list of the **most widespread and critical errors** that can lead to serious vulnerabilities in software. They are often easy to find, and easy to exploit. They are dangerous because they will **frequently allow attackers to completely take over the software**, steal data, or prevent the software from working at all.*

How critical are memory corruption bugs?

MITRE ranking [http://cwe.mitre.org/top25/]:

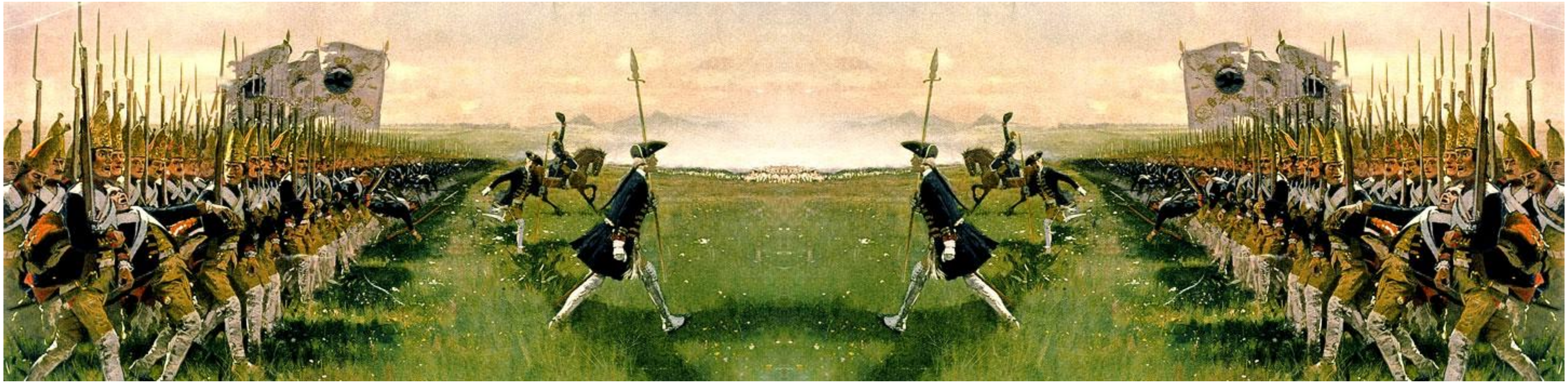
*The 2011 CWE/SANS Top 25 Most Dangerous Software Errors is a list of the **most widespread and critical errors** that can lead to serious vulnerabilities in software. They are often easy to find, and easy to exploit. They are dangerous because they will **frequently allow attackers to completely take over the software**, steal data, or prevent the software from working at all.*

memory corruption bugs are considered
one of the **top three** most dangerous software errors

“Eternal War in Memory”

defensive research

offensive research



new protections

new attacks

Exemplary exploits: how-to

Use Return Oriented Programming

Defend against attacks

Define safety

Data Execution Prevention

The idea:

mark areas of memory as either "executable" or "non executable"

`r-x`	code pages
`rw-`	data pages (stack, heap)
`(r|-)wx`	must never happen

Let’s check:

```
cat /proc/<PID>/maps
```

Data Execution Prevention

```
cat /proc/<PID>/maps
```

```
00400000-00401000 r-xp 00000000 08:02 15075435                   /home/i.rub/ROP/waiter
00600000-00601000 r--p 00000000 08:02 15075435                   /home/i.rub/ROP/waiter
00601000-00602000 rw-p 00001000 08:02 15075435                   /home/i.rub/ROP/waiter
01977000-019a9000 rw-p 00000000 00:00 0                          [heap]
7ff302b76000-7ff302b8c000 r-xp 00000000 08:01 4214577            /lib/x86_64-linux-gnu/libgcc_s.so.1
7ff302b8c000-7ff302d8b000 ---p 00016000 08:01 4214577            /lib/x86_64-linux-gnu/libgcc_s.so.1
7ff302d8b000-7ff302d8c000 r--p 00015000 08:01 4214577            /lib/x86_64-linux-gnu/libgcc_s.so.1
...
7ff30398a000-7ff30398b000 rw-p 00000000 00:00 0
7fff65113000-7fff65135000 rw-p 00000000 00:00 0                  [stack]
7fff6516a000-7fff6516c000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

We already know how to overcome this protection.

Address Space Layout Randomization

The idea:

each process’ address space is randomized

so that stack, heap and libraries are mapped to some random address.

Address Space Layout Randomization

The idea:

each process' address space is randomized

so that stack, heap and libraries are mapped to some random address.

Let's check:

```
cat /proc/<PID>/maps
```

```
ldd a.out
```

`[vsyscall]` is at a fixed address...

Address Space Layout Randomization

The idea:

each process’ address space is randomized

so that stack, heap and libraries are mapped to some **“random”** address.

Let’s check:

```
cat /proc/<PID>/maps
```

```
ldd a.out
```

`[vsyscall]` is at a fixed address…

How “random” was* ASLR?

* fixed in Linux 2.6.30

```
/*
 * Get a random word for internal kernel use only.[...]
 */
unsigned int get_random_int(void)
{
        /*
         * Use IP's RNG. It suits our purpose perfectly: it re-keys itself
         * every second, from the entropy pool (and thus creates a limited
         * drain on it), and uses halfMD4Transform within the second. We
         * also mix it with jiffies and the PID:
         */
        return secure_ip_id((__force __be32)(current->pid + jiffies));
}
```

`secure_ip_id(x)` is a PRF depending solely on argument **x** and the **key**, which changes every 5 minutes (not `every second’)

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-whitepaper.pdf

How “random” was* ASLR?

* fixed in Linux 2.6.30

within 5 minutes `get_random_int()` depends solely on **(jiffies + pid)**

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

How “random” was* ASLR?

* fixed in Linux 2.6.30

within 5 minutes `get_random_int()` depends solely on **(jiffies + pid)**

jiffies’ granularity is known (e.g. 4ms for Linux 2.6.13+, on Intel x86)

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

How “random” was* ASLR?

* fixed in Linux 2.6.30

within 5 minutes `get_random_int()` depends solely on **(jiffies + pid)**

jiffies’ granularity is known (e.g. 4ms for Linux 2.6.13+, on Intel x86)

We can recreate conditions to get exactly the same “random” value:

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

How “random” was* ASLR?

* fixed in Linux 2.6.30

within 5 minutes `get_random_int()` depends solely on **(jiffies + pid)**

jiffies’ granularity is known (e.g. 4ms for Linux 2.6.13+, on Intel x86)

We can recreate conditions to get exactly the same “random” value:

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

How “random” was* ASLR?

* fixed in Linux 2.6.30

within 5 minutes `get_random_int()` depends solely on **(jiffies + pid)**

jiffies’ granularity is known (e.g. 4ms for Linux 2.6.13+, on Intel x86)

We can recreate conditions to get exactly the same “random” value:

Timeframe for attack: 32768 × 4ms = 131s = **2min 11s**

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

How common is ASLR?

In order to benefit from ASLR protection an executable has to be compiled as position independent executable (PIE).

http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/

How common is ASLR?

In order to benefit from ASLR protection an executable has to be compiled as position independent executable (PIE).

A tool, **Checksec**, was applied to verify if binaries used security mechanisms:

Distribution	**all** binaries	**PIE** binaries		**not PIE** binaries	
Ubuntu 12.10	646	111	**17%**	535	**83%**
Debian 6	592	71	**10%**	531	**90%**
CentOS 6.3	1340	217	**16%**	1123	**84%**

http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/

Playing with ASLR at home

Setting temporary level of randomization:

```
echo 0  | sudo tee /proc/sys/kernel/randomize_va_space
```

Levels of randomization:

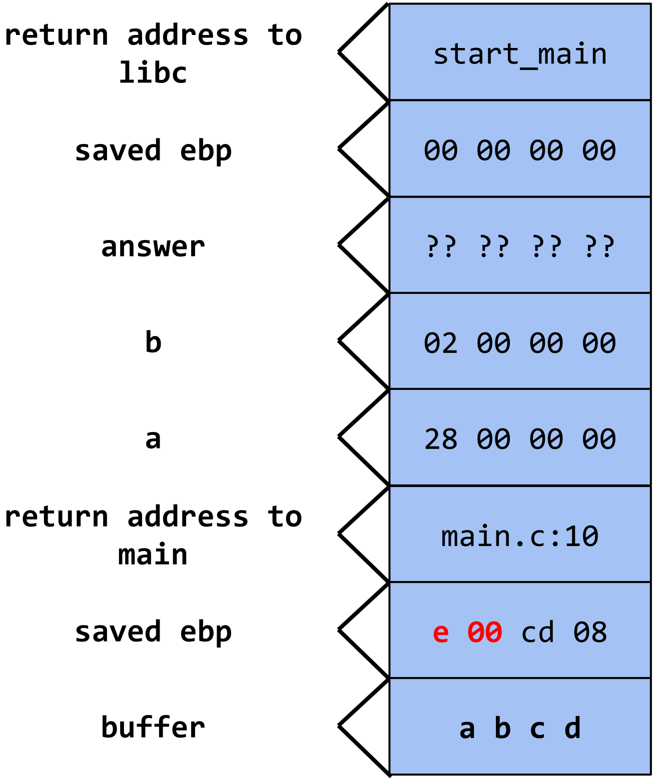
0	**No randomization**	Everything is static
1	**Conservative randomization**	Shared libraries, stack, mmap(), VDSO and heap are randomized
2	**Full randomization**	Memory managed through brk() is also randomized

Canaries

a canary in the coal mine

Danger detected!

```
int foo(int a, int b)
{
  char buffer[4] = "abcde";
  return 42;
}

int main(int argc, char** argv)
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```



Danger detected!

```
int foo(int a, int b)
{
  char buffer[4] = "abcde";
  return 42;
}

int main(int argc, char** argv)
{
  int answer;
  answer = add(40, 2);
  return 0;
}
```

return address to libc	start_main
saved ebp	00 00 00 00
answer	?? ?? ?? ??
b	02 00 00 00
a	28 00 00 00
return address to main	main.c:10
saved ebp	e 00 cd 08
buffer	a b c d

Random magic value is inserted next to saved ebp
and verified afterwards before registers update.

Let’s not overwrite the canary

If we are still able to overwrite local values…

pointer subterfuge

http://blogs.msdn.com/b/michael_howard/archive/2006/01/30/520200.aspx

Let’s not overwrite the canary

If we are still able to overwrite local values…

pointer subterfuge

```
void SomeFunc() {
  // do something
}

typedef void (*FUNC_PTR )(void);
```

http://blogs.msdn.com/b/michael_howard/archive/2006/01/30/520200.aspx

Let’s not overwrite the canary

If we are still able to overwrite local values…

pointer subterfuge

```
void SomeFunc() {
  // do something
}

typedef void (*FUNC_PTR )(void);
```

```
int DangerousFunc(char *szString) {
  char buf[32];
  strcpy(buf,szString);
  FUNC_PTR fp = (FUNC_PTR)(&SomeFunc);
  // Other code
  (*fp)();
  return 0;
}
```

http://blogs.msdn.com/b/michael_howard/archive/2006/01/30/520200.aspx

Let’s not overwrite the canary

If we are still able to overwrite local values...

pointer subterfuge

```c
void SomeFunc() {
  // do something
}

typedef void (*FUNC_PTR )(void);
```

```c
int DangerousFunc(char *szString) {
  char buf[32];
  strcpy(buf,szString);
  FUNC_PTR fp = (FUNC_PTR)(&SomeFunc);
  // Other code
  (*fp)();
  return 0;
}
```

http://blogs.msdn.com/b/michael_howard/archive/2006/01/30/520200.aspx

How to get around?

How to get around?

Overwriting the master-canary?

It is stored at a static location. If there is no ASLR.

How to get around?

Overwriting the master-canary?

It is stored at a static location. If there is no ASLR.

Guessing the canary value?

If people care for performance rather than security - yes.

How to get around?

Overwriting the master-canary?

It is stored at a static location. If there is no ASLR.

Guessing the canary value?

If people care for performance rather than security - yes.

`ENABLE_STACKGUARD_RANDOMIZE` is actually off on most architectures, canary defaults to `0xff0a000000000000`.

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

Offense-defense summary

`DEP`	easy
`ASLR`	feasible
`canaries`	depends*
`DEP + ASLR`	feasible
`DEP + canaries`	depends*
`ASLR + canaries`	hard
`DEP + ASLR + canaries`	hard

*depends on environmental factors or certain code flaws

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

How common are memory corruption bugs?

Pwn2Own 2012:

Google Chrome sandbox exploited for the first time!

VUPEN team used a pair of zero-day vulnerabilities to take complete control of a fully patched 64-bit Windows 7.

http://www.zdnet.com/article/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/

How common are memory corruption bugs?

Pwn2Own 2012:

Google Chrome sandbox exploited for the first time!

VUPEN team used a pair of zero-day vulnerabilities to take complete control of a fully patched 64-bit Windows 7.

Pwnium 2012:

Sergey Glazunov and “PinkiePie” each prepared exploits for Chrome.

Google issued a fix to Chrome users in less than 24 hours after the Pwnium exploits were demonstrated.

http://www.zdnet.com/article/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/

How VUPEN owned the system?

*We had to use **two vulnerabilities**. The first one was to bypass DEP and ASLR on Windows and a second one to break out of the Chrome sandbox.*

Chaouki Bekrar (VUPEN co-founder)

http://www.zdnet.com/article/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/

How VUPEN owned the system?

*We had to use **two vulnerabilities**. The first one was to bypass DEP and ASLR on Windows and a second one to break out of the Chrome sandbox.*

*It was a **use-after-free** vulnerability in the default installation of Chrome.*

Chaouki Bekrar (VUPEN co-founder)

http://www.zdnet.com/article/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/

How VUPEN owned the system?

*We had to use **two vulnerabilities**. The first one was to bypass DEP and ASLR on Windows and a second one to break out of the Chrome sandbox.*

*It was a **use-after-free** vulnerability in the default installation of Chrome.*

*This just shows that **any browser, or any software, can be hacked** if there is enough motivation and skill.*

Chaouki Bekrar (VUPEN co-founder)

http://www.zdnet.com/article/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/

References

SoK: Eternal War in Memory:
L.Szekeres (Stony Brook University), M.Payer (University of California, Berkeley), T.Wei (Peking University), D.Song, (University of California, Berkeley), 2103

http://www.cs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf

Stack Smashing as of Today:
A State-of-the-Art Overview on Buffer Overflow Protections on linux_x86_64
Hagen Fritsch, Technische Universität München, Black Hat Europe – Amsterdam, 2009

https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

Exemplary exploits: how-to

Use Return Oriented Programming

Defend against attacks

Define safety

When is our software safe?

Is it possible to...

...claim that a program is resistant to software attacks?

...create applications in a safe language?

...enforce safety in case of C/C++?

When is our software safe?

Is it possible to...

...claim that a program is resistant to software attacks?

...create applications in a safe language?

...enforce safety in case of C/C++?

When is our software safe?

Is it possible to...

...claim that a program is resistant to software attacks?

...create applications in a safe language?

...enforce safety in case of C/C++?

Memory safety

all possible **executions** are memory safe

Memory safety

all possible **executions** are memory safe

⬇

a **program** is memory safe

Memory safety

all possible **executions** are memory safe

a **program** is memory safe

all possible **programs** are memory safe

Memory safety

all possible **executions** are memory safe

a **program** is memory safe

all possible **programs** are memory safe

a **programming language** is memory safe

Memory-safe execution

Intuitively, it means that none of these bad things happen:

Memory-safe execution

Intuitively, it means that none of these bad things happen:

null pointer dereference

use of uninitialized memory

Memory-safe execution

Intuitively, it means that none of these bad things happen:

null pointer dereference

use of uninitialized memory

use after free

illegal free (of an already-freed pointer, or a non-malloced pointer)

Memory-safe execution

Intuitively, it means that none of these bad things happen:

null pointer dereference

use of uninitialized memory

use after free

illegal free (of an already-freed pointer, or a non-malloced pointer)

buffer overflow

Memory-safe execution

***Defined memory*:**

Memory-safe execution

***Defined memory*:**

allocated **on the heap** (malloc)

Memory-safe execution

***Defined memory*:**

allocated **on the heap** (malloc)

allocated **on the stack** (local variables, function parameters)

Memory-safe execution

***Defined memory*:**

allocated **on the heap** (malloc)

allocated **on the stack** (local variables, function parameters)

global variables in **static data** area

Memory-safe execution

***Defined memory*:**

allocated **on the heap** (malloc)

allocated **on the stack** (local variables, function parameters)

global variables in **static data** area

In a memory-safe execution *undefined memory* cannot be accessed.

Memory-safe execution

Intuitively, it means that none of these bad things happen:

null pointer dereference

use of uninitialized memory

use after free

illegal free (of an already-freed pointer, or a non-malloced pointer)

buffer overflow

Memory-safe execution

Intuitively, it means that none of these bad things happen:

~~null pointer dereference~~

use of uninitialized memory

use after free

illegal free (of an already-freed pointer, or a non-malloced pointer)

buffer overflow

Memory-safe execution

Intuitively, it means that none of these bad things happen:

~~null pointer dereference~~

~~use of uninitialized memory~~

use after free

illegal free (of an already-freed pointer, or a non-malloced pointer)

buffer overflow

Memory-safe execution

Intuitively, it means that none of these bad things happen:

~~null pointer dereference~~

~~use of uninitialized memory~~

~~use after free~~

illegal free (of an already-freed pointer, or a non-malloced pointer)

buffer overflow

Memory-safe execution

Intuitively, it means that none of these bad things happen:

~~null pointer dereference~~

~~use of uninitialized memory~~

~~use after free~~

~~illegal free (of an already-freed pointer, or a non-malloced pointer)~~

buffer overflow

Memory-safe execution

Still there is a problem with buffer overflow:

```c
int x;
int buf[4];
buf[4] = 3; /* overwrites x */
```

Memory-safe execution

Still there is a problem with buffer overflow:

```
int x;
int buf[4];
buf[4] = 3; /* overwrites x */
```

Let's add to the definition: *infinite spacing*.

We assume that memory regions are allocated infinitely far apart.

Memory-safe execution

Still there is a problem with buffer overflow:

```
struct foo {
  int buf[4];
  int x;
};
struct foo *pf = malloc(sizeof(struct foo));
pf->buf[4] = 3; /* overwrites pf->x */
```

Memory-safe execution

Still there is a problem with buffer overflow:

```c
struct foo {
  int buf[4];
  int x;
};
struct foo *pf = malloc(sizeof(struct foo));
pf->buf[4] = 3; /* overwrites pf->x */
```

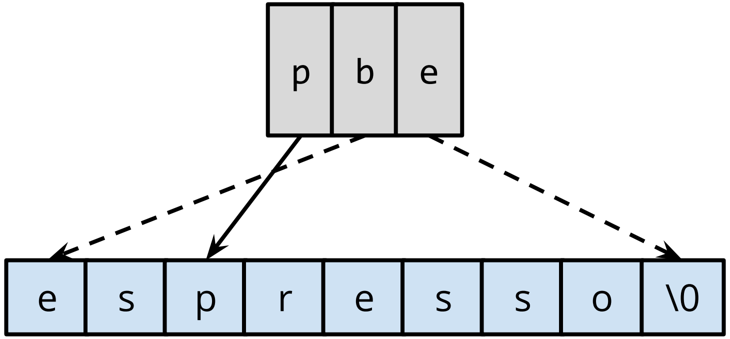
Should we assume *infinite spacing* between structure fields?

Better not.

Fat pointers ensure spatial safety

Each pointer consists of three elements: `(p, b, e)`.

```
char *pc = “espresso”;
pc += 3;
```



How to enforce memory safety?

HardBound:
Architectural Support for Spatial Safety of the C Programming Language

http://www.cis.upenn.edu/acg/papers/asplos08_hardbound.pdf

SoftBound:
Highly Compatible and Complete Spatial Memory Safety for C

http://llvm.org/pubs/2009-06-PLDI-SoftBound.pdf

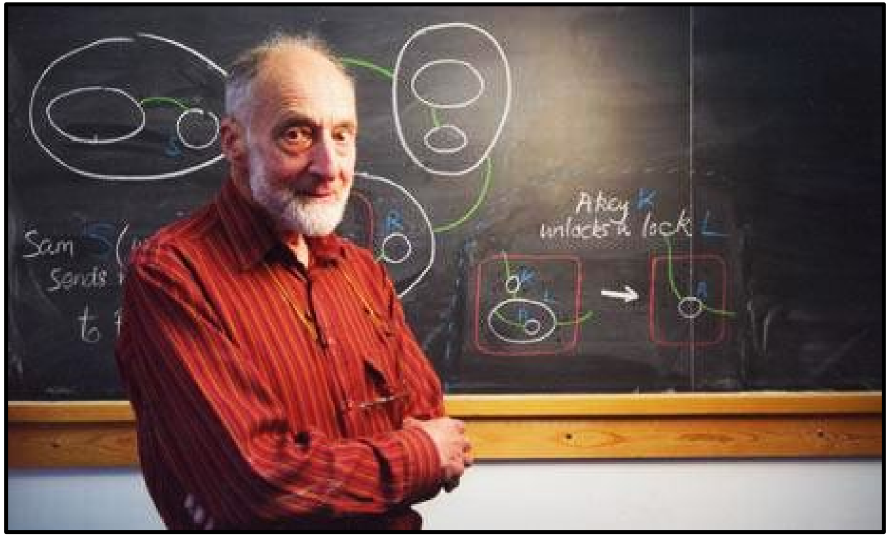
DieHard:
Probabilistic Memory Safety for Unsafe Languages

https://people.cs.umass.edu/~emery/pubs/fp014-berger.pdf

Type safety

A Theory of Type Polymorphism in Programming: Robin Milner, 1978

“***Well typed*** *programs* ***cannot go wrong***.”



Syntax vs. semantics

“Colorless green ideals sleep furiously.”

```
{
    char buf[4];
    buf[4] = ‘x’;
}
```

Type-safe language

In a type-safe language:

the language’s type system ensures that
syntactically correct programs are **well defined**.

Examples:

Java, C#

Python, Ruby

Extensions to type systems

Some programs are well defined but incorrect in a given type system:

```
if (p) x = 5;
  else x = "hello";
if (p) return x + 5;
  else return strlen(x);
```

Extensions to type systems

Some programs are well defined but incorrect in a given type system:

```
if (p) x = 5;
  else x = "hello";
if (p) return x + 5;
  else return strlen(x);
```

Types could carry much more information expressed as invariants:

```
{v: int | 0 <= v}
```

```
{v: int | v % 2}
```

References

What is memory safety?
The Programming Languages Enthusiast, July 2014

http://www.pl-enthusiast.net/2014/07/21/memory-safety/

What is type safety?
The Programming Languages Enthusiast, August 2014

http://www.pl-enthusiast.net/2014/08/05/type-safety/

Let’s talk about:

- Bases of memory management for a single process
- Exemplary exploits (Return Oriented Programming)
- **LLVM tools that prevents some of software attacks**

LLVM tools

Why LLVM?

Sanitizers

Work in progress

How did it start?

Low Level Virtual Machine project starts in 2000 at the University of Illinois

How did it start?

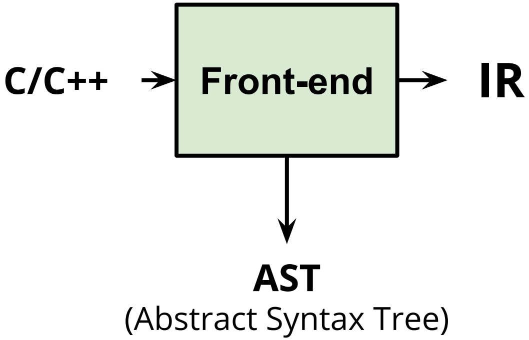
Low Level Virtual Machine project starts in 2000 at the University of Illinois

The main concept includes:

a modular architecture

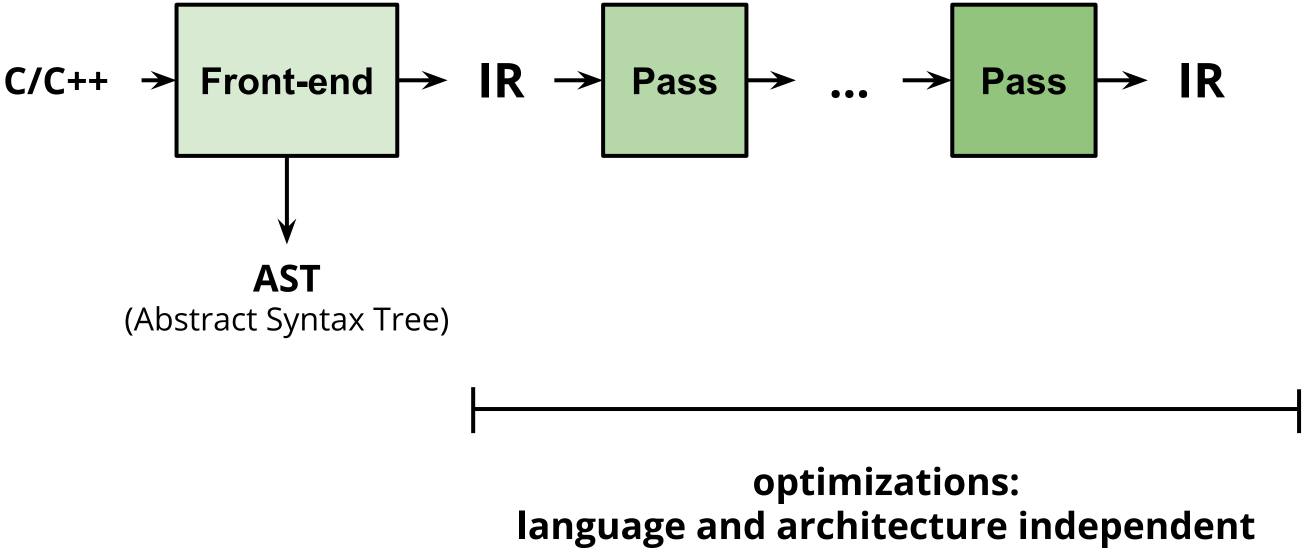
LLVM architecture

LLVM was designed as a set of reusable libraries with well-defined interfaces.



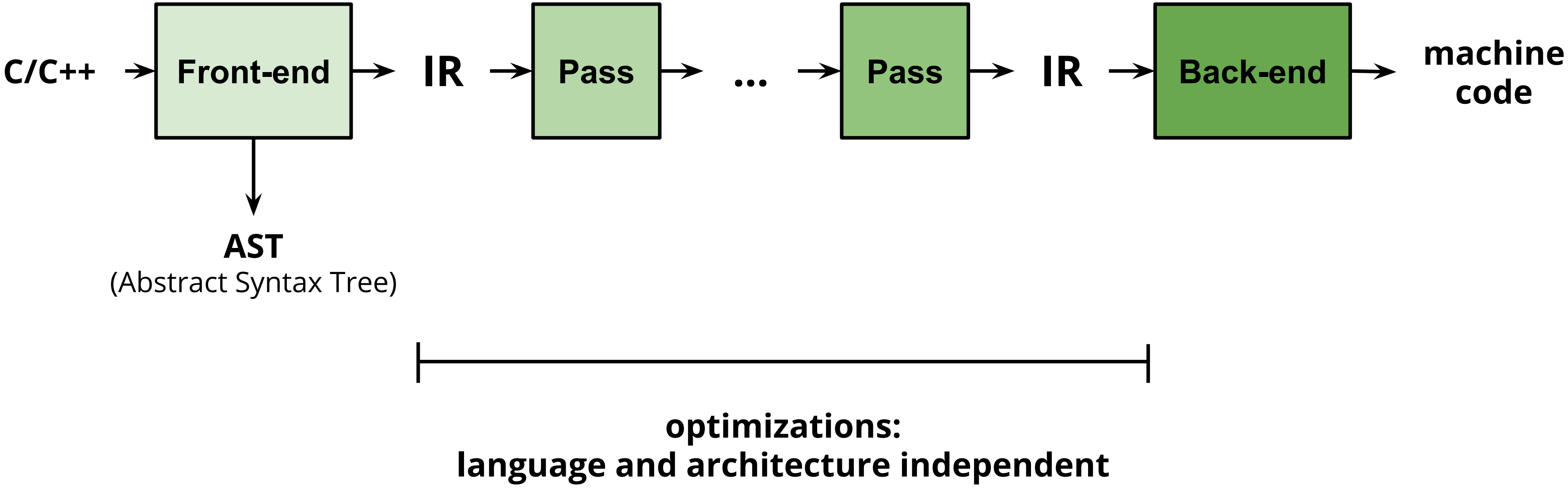
LLVM architecture

LLVM was designed as a set of reusable libraries with well-defined interfaces.



LLVM architecture

LLVM was designed as a set of reusable libraries with well-defined interfaces.



How did it start?

Low Level Virtual Machine project starts in 2000 at the University of Illinois

The main concept includes:

a modular architecture

an intermediary code representation: **IR**

SSA form

How did it start?

Low Level Virtual Machine project starts in 2000 at the University of Illinois

The main concept includes:

a modular architecture

an intermediary code representation: **IR**

SSA form

Purpose:

a `hackable and hacking' compiler

Who uses LLVM and why?

LLVM is heavily used in **both academia and industry**

especially: in work targeted at **high-performance computing**

Who uses LLVM and why?

LLVM is heavily used in **both academia and industry**

especially: in work targeted at **high-performance computing**

Portable Computing Language (pocl):
an open source implementation of the OpenCL

Who uses LLVM and why?

LLVM is heavily used in **both academia and industry**

especially: in work targeted at **high-performance computing**

Portable Computing Language (pocl):
an open source implementation of the OpenCL

Apple Inc.

Adobe

Who uses LLVM and why?

LLVM is heavily used in **both academia and industry**

especially: in work targeted at **high-performance computing**

Portable Computing Language (pocl):
an open source implementation of the OpenCL

Apple Inc.

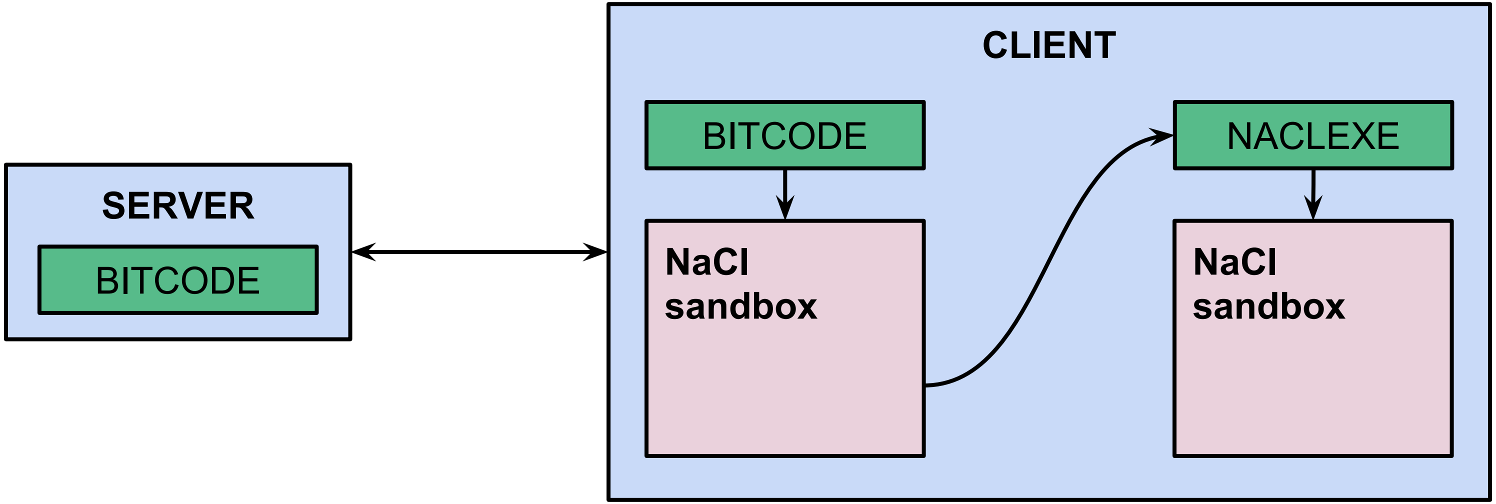
Adobe

...

PNaCl

PNaCl for Chrome

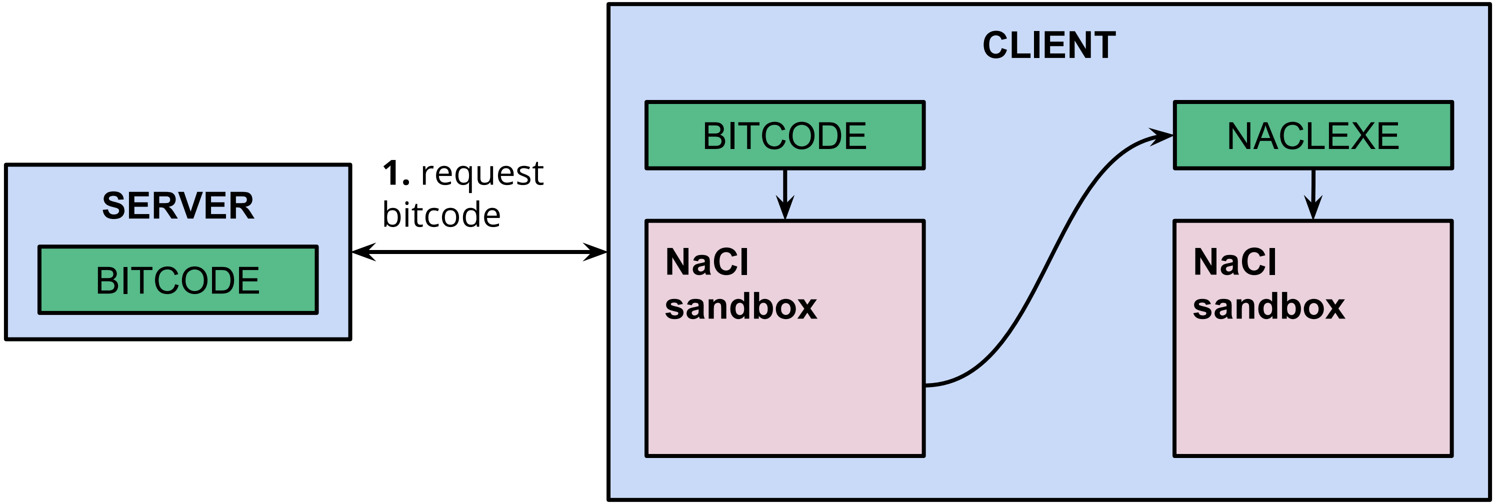
*PNaCl introduces a twist in the toolchain: **instead of compiling C/C++ applications for each of the hardware platforms targeted**, developers now need to **generate a single LLVM bitcode** which is them loaded by any Chrome client and translated to native code, validated and executed locally.*



http://www.infoq.com/news/2013/05/pnacl-google-chrome-llvm

PNaCl for Chrome

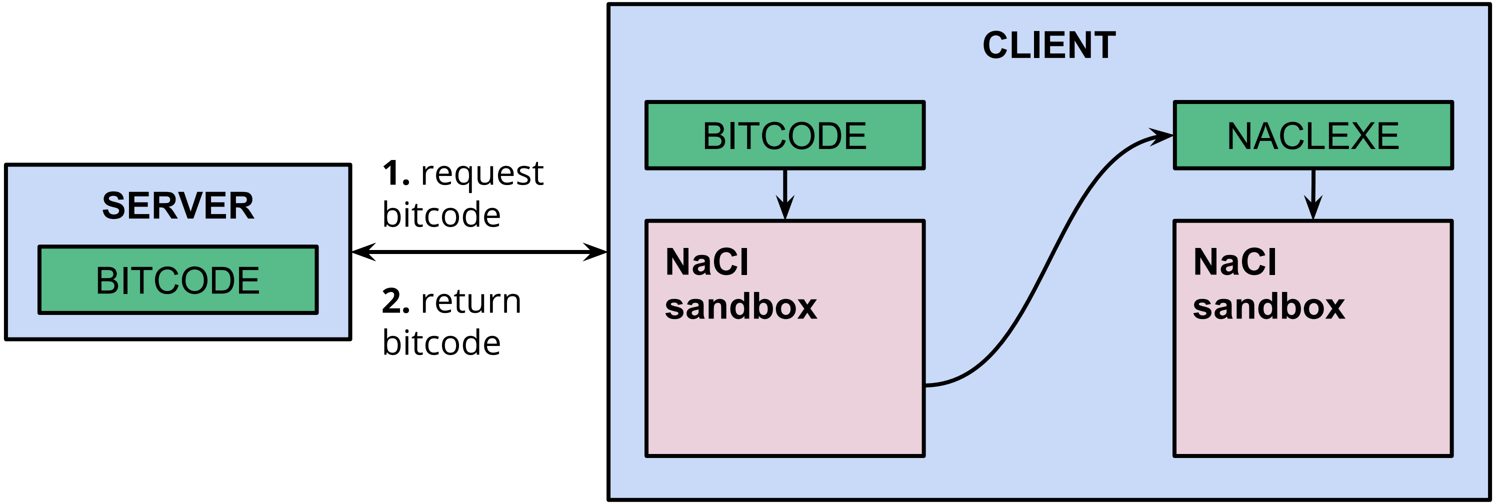
*PNaCl introduces a twist in the toolchain: **instead of compiling C/C++ applications for each of the hardware platforms targeted**, developers now need to **generate a single LLVM bitcode** which is them loaded by any Chrome client and translated to native code, validated and executed locally.*



http://www.infoq.com/news/2013/05/pnacl-google-chrome-llvm

PNaCl for Chrome

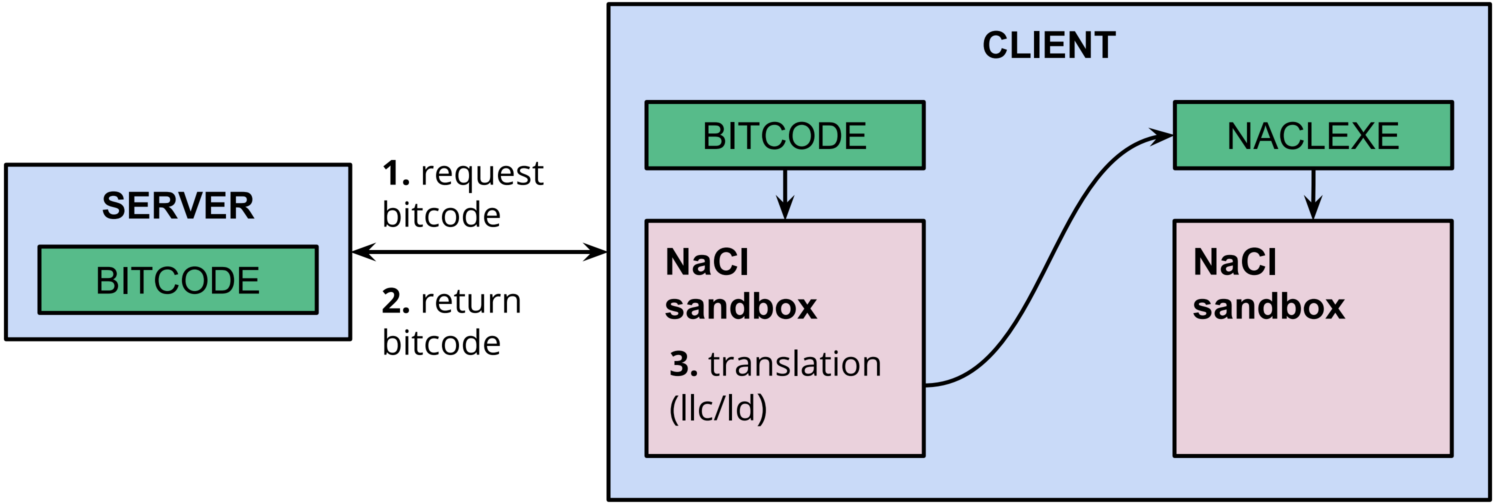
*PNaCl introduces a twist in the toolchain: **instead of compiling C/C++ applications for each of the hardware platforms targeted**, developers now need to **generate a single LLVM bitcode** which is them loaded by any Chrome client and translated to native code, validated and executed locally.*



http://www.infoq.com/news/2013/05/pnacl-google-chrome-llvm

PNaCl for Chrome

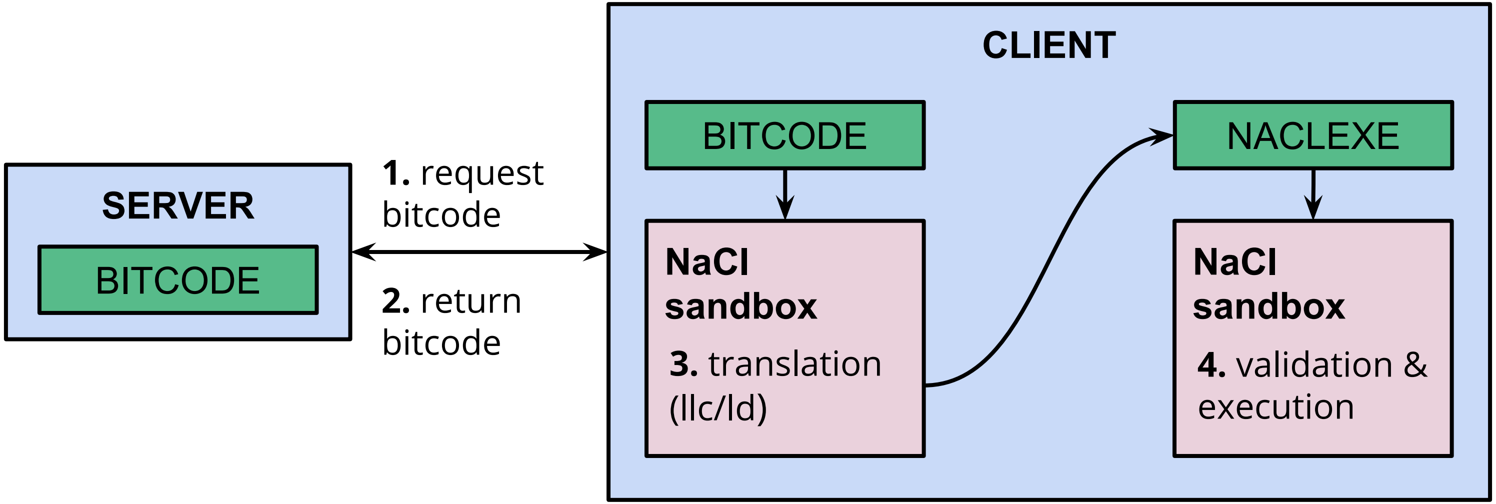
*PNaCl introduces a twist in the toolchain: **instead of compiling C/C++ applications for each of the hardware platforms targeted**, developers now need to **generate a single LLVM bitcode** which is them loaded by any Chrome client and translated to native code, validated and executed locally.*



http://www.infoq.com/news/2013/05/pnacl-google-chrome-llvm

PNaCl for Chrome

*PNaCl introduces a twist in the toolchain: **instead of compiling C/C++ applications for each of the hardware platforms targeted**, developers now need to **generate a single LLVM bitcode** which is them loaded by any Chrome client and translated to native code, validated and executed locally.*



http://www.infoq.com/news/2013/05/pnacl-google-chrome-llvm

Where is LLVM going?

Low Level Virtual Machine project starts in 2000 at the University of Illinois

the umbrella project that includes a variety of compiler and low-level tool technologies

Where is LLVM going?

Low Level Virtual Machine project starts in 2000 at the University of Illinois

the umbrella project that includes a variety of compiler and low-level tool technologies

LLVM core (an optimizer and a code generator)

clang - a C/C++ compiler

OpenMP, polly, klee...

LLVM tools

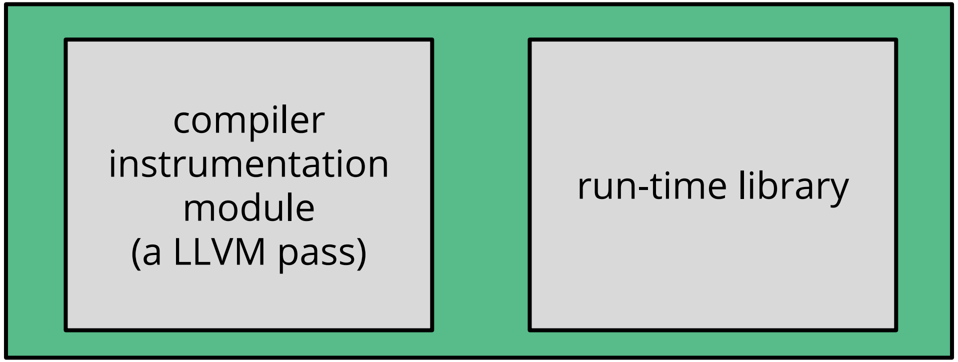
Why LLVM?

Sanitizers

Work in progress

MemorySanitizer - MSan

is a detector of **uninitialized reads** that affect program execution




```
%clang -fsanitize=memory -fPIE -pie -fno-omit-frame-pointer -g -O1 ex.cc
```

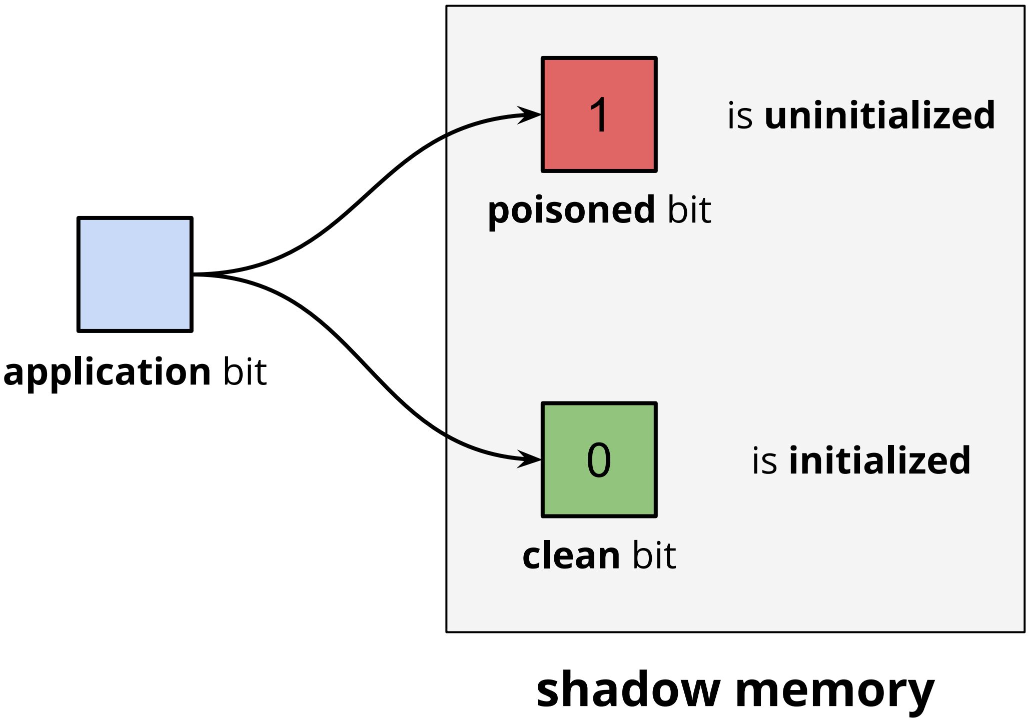
MemorySanitizer in action

```
#include <stdio.h>

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

 int* a = new int[10];
 a[2] = 2;
 printf("ARGC: %d\n", argc);
 int b = a[argc];
 if (b)
     printf("HERE\n");
 return 0;

}
```

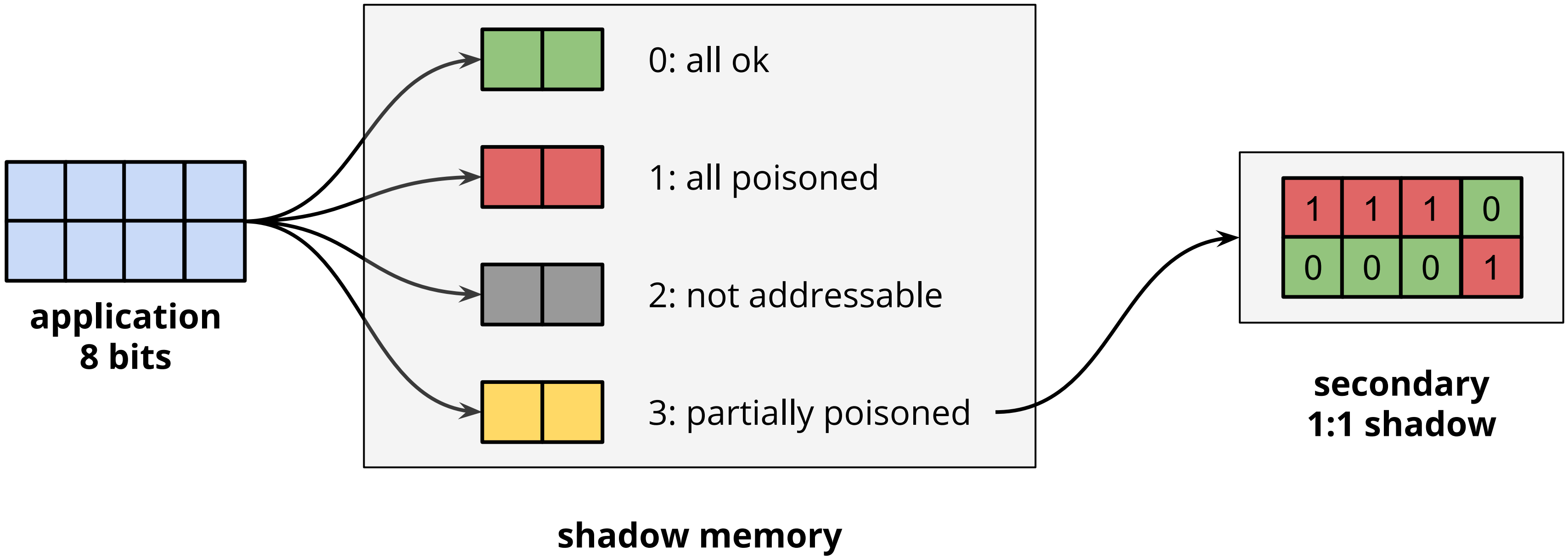
```
MSAN_SYMBOLIZER_PATH=$(which llvm-symbolizer-3.4) ./a.out
```

```
ARGC: 1
==4312== WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7f1e6d703d21 in main (/home/i.rub/ROP/MSAN/a.out+0x73d21)
#1 0x7f1e6c272ec4 (/lib/x86_64-linux-gnu/libc.so.6+0x21ec4)
#2 0x7f1e6d7039dc in _start (/home/i.rub/ROP/MSAN/a.
out+0x739dc)

SUMMARY: MemorySanitizer: use-of-uninitialized-value ??:0 main
Exiting
```

MemorySanitizer - the idea



Shadow memory in Memcheck



Shadow memory in Memcheck

Motivation:

Partially defined bytes are rarely involved in more than 0.1% of memory accesses, and are not present at all in many programs.

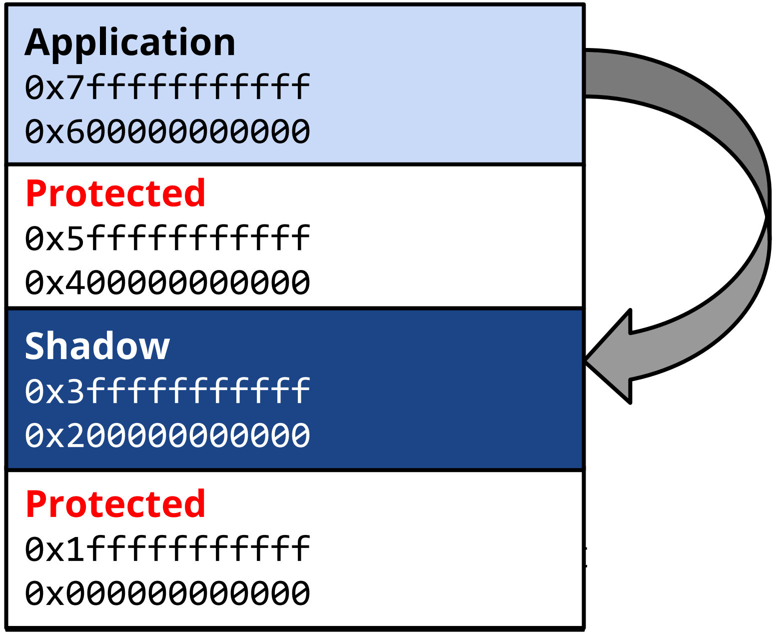
How to Shadow Every Byte of Memory Used by a Program, Nicholas Nethercote, Julian Seward

But still, the Valgrind tool is:

slower

prone to racy updates on multiprocessor machines

Direct 1:1 shadow mapping in MSan




```
shadow = addr - 0x400000000000
```

When to report errors?

We do not want to report every load of uninitialized data:

```
struct foo {
  char x;
  // 3-byte padding
  int y;
};
```

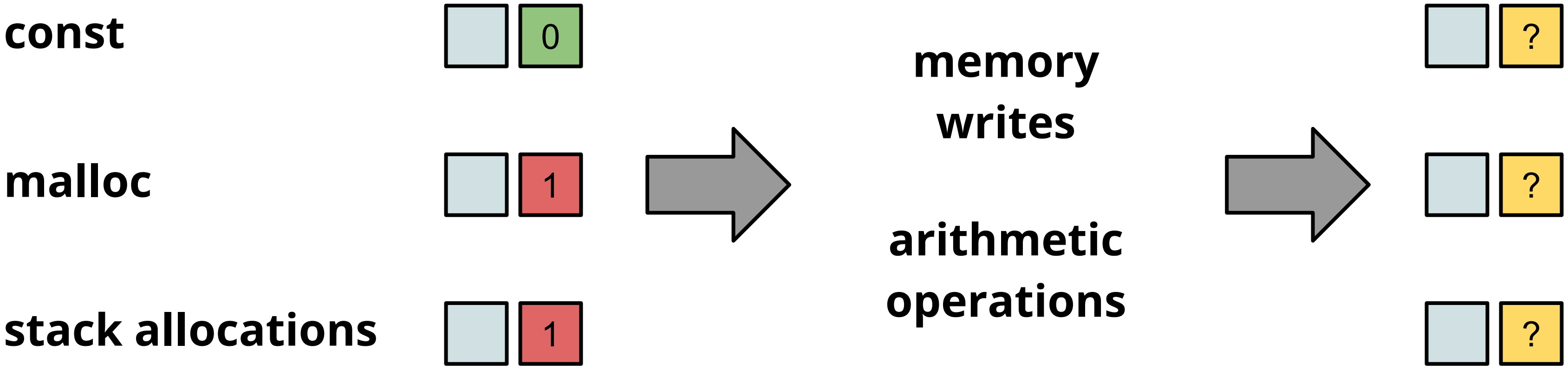
Also, it is OK to **copy** uninitialized data round.

Calculations on such data are OK too, as long as the result is discarded.

UMR are reported in case of: branches, syscalls, pointer dereferences.

Shadow propagation

Shadow memory is assigned to every value from the very beginning.



Shadow is unpoisoned when constants are stored.

Shadow propagation

How to pass shadow information through expressions?

Let **`A`** be a value and **`A'`** - the shadow.

For each **op** we have to define **`op'`**:

```
A   = op B, C
A'  = op' B, C, B', C'
```

Shadow propagation

How to pass shadow information through expressions?

Let **`A`** be a value and **`A'`**- the shadow.

For each **op** we have to define **`op'`**:

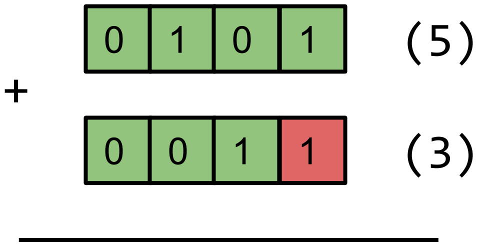
```
A   = op B, C
A'  = op' B, C, B', C'
```

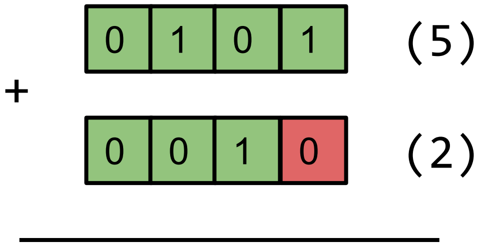
<u>*Example*:</u>

```
A = B xor C: A' = B' | C'
A = B & C: A' = (B' & C') | (B & C') | (B' & C)
```

Difficulties

It is not always possible to efficiently implement op ' .



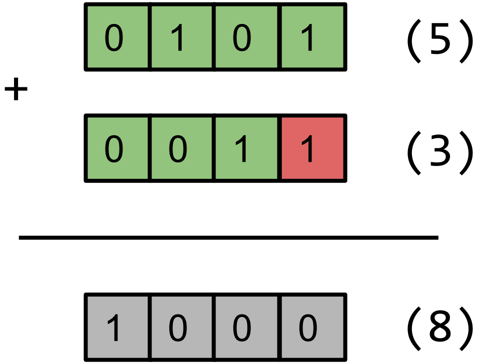


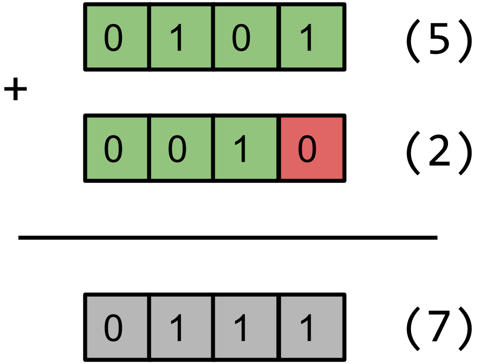
Difficulties

It is not always possible to efficiently implement op' .

Difficulties

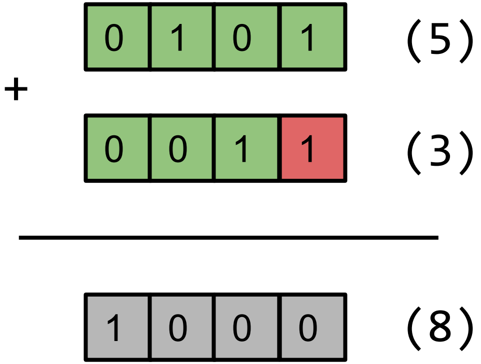
It is not always possible to efficiently implement op' **.**

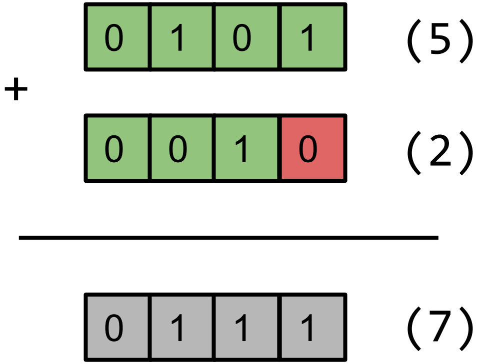




Difficulties

It is not always possible to efficiently implement `op'`.





Often, an **approximated propagation** of shadow is used.

```
A = B + C: A' = B' | C'
```

Difficulties

MSan is one of last passes - it operates on a strongly optimized IR.

```
struct S {
  int a : 3;
  int b : 5;
};

bool f(S *s) { return s->b; }
```

Difficulties

MSan is one of last passes - it operates on a strongly optimized IR.

```
struct S {
  int a : 3;
  int b : 5;
};

bool f(S *s) { return s->b; }
```

```
*( unsigned char *)s > 7
```

Difficulties

MSan is one of last passes - it operates on a strongly optimized IR.

```
struct S {
  int a : 3;
  int b : 5;
};

bool f(S *s) { return s->b; }
```

→

```
*( unsigned char *)s > 7
```

If all relational comparisons are instrumented correctly then benchmarks show slowdown of up to 50%.

Difficulties

Missing any write instruction causes false reports.

ALL stores in the program must be monitored, including stores in libc, libstdc++, syscalls...

Solutions:

recompiled and instrumented libc, libc++

wrappers for common libc functions

DynamoRIO (MSanDr tool) - for binary instrumentation

Difficulties

Where was the poisoned memory allocated?

```
a = malloc( … );
…
b = malloc( … );
…
c = *a + *b;
if ( c ) … // reported error
```

Is **a** guilty or **b**?

Origin tracking

We have to allocate additional 4 bytes to keep the origin ID.

Additional slowdown: **2x (total: 6x)**

RAM: **3x + malloc stack traces**

Origin tracking

We have to allocate additional 4 bytes to keep the origin ID.

Additional slowdown: **2x (total: 6x)**

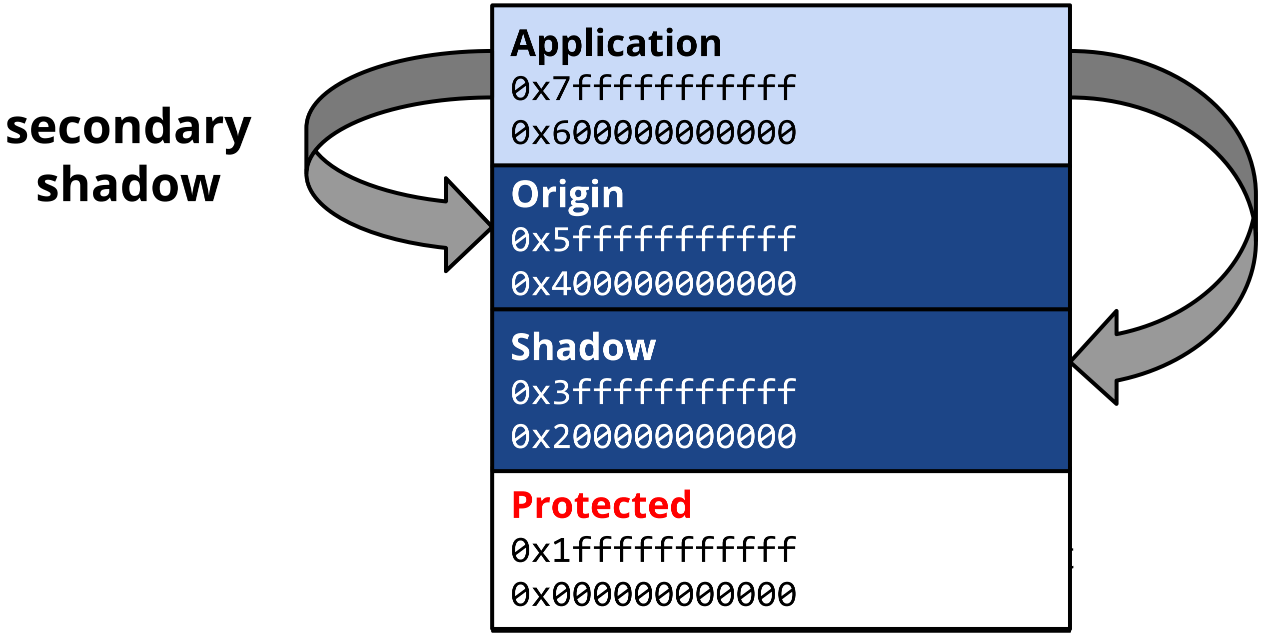
RAM: **3x + malloc stack traces**

Example:

Let **`A`** be a value, **`A'`**- the shadow, **`A''`** - the origin.

```
A = op B, C, D

A'' = (D') ? (D'') : (C' ? C'' : B'')
```

Direct 1:1 shadow mapping




```
shadow = addr - 0x400000000000
origin = addr - 0x200000000000
```

Advanced origin tracking

In this mode MSan prints **stack traces of all memory stores** along the path: from the allocation the use of the uninitialized value.

Advanced origin tracking

In this mode MSan prints **stack traces of all memory stores** along the path: from the allocation the use of the uninitialized value.

There is a **hash map of origin IDs** (each store operation has its entry):

(previous origin ID, stack trace) $\longrightarrow$ new origin ID

Origin ID is a descriptor of a sequence of undefined stores starting with its creation.

Advanced origin tracking

In this mode MSan prints **stack traces of all memory stores** along the path: from the allocation the use of the uninitialized value.

There is a **hash map of origin IDs** (each store operation has its entry):

(previous origin ID, stack trace) ⟶ new origin ID

Origin ID is a descriptor of a sequence of undefined stores starting with its creation.

But: some **limits** for the size of the tracked history have to be set.

Advanced origin tracking

`B = store A`

```
B'' = look up the value of
(A'', current stack trace)
      in the hash map
```

Advanced origin tracking

B = store A

B'' = look up the value of
(A'', current stack trace)
in the **hash map**

if (B'')

YES

return B''

Advanced origin tracking

B = store A

B'' = look up the value of (A'', current stack trace) in the **hash map**

if (B'')

NO

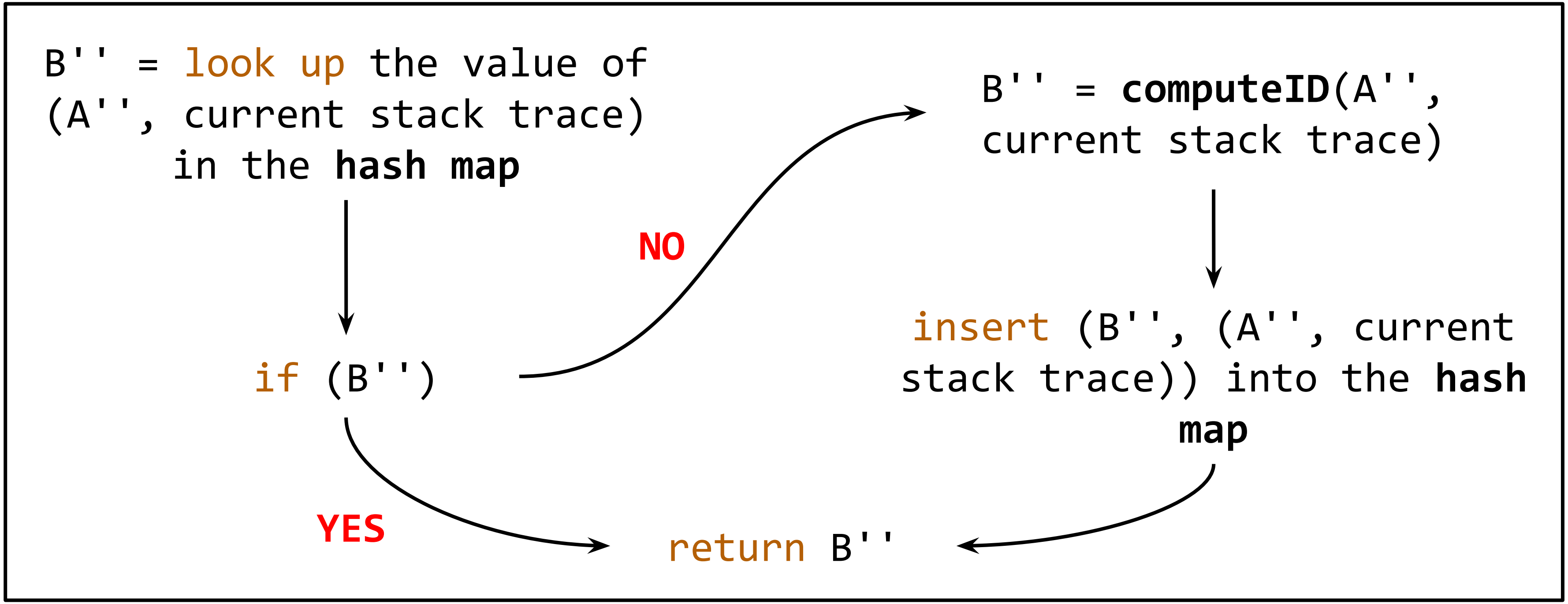
B'' = **computeID**(A'', current stack trace)

YES

return B''

Advanced origin tracking

B = store A



Is MSan superior over Memcheck?

Is **compiler instrumentation** superior over **binary instrumentation**?

IR carries more information, thus:

Is MSan superior over Memcheck?

Is **compiler instrumentation** superior over **binary instrumentation**?

IR carries more information, thus:

faster instrumentation

Is MSan superior over Memcheck?

Is **compiler instrumentation** superior over **binary instrumentation**?

IR carries more information, thus:

faster instrumentation

less false positives (e.g. in case of lazy computations)

Is MSan superior over Memcheck?

Is **compiler instrumentation** superior over **binary instrumentation**?

IR carries more information, thus:

faster instrumentation

less false positives (e.g. in case of lazy computations)

all the names of local variables are known

Is MSan superior over Memcheck?

Used for: **Chrome**, **LLVM**

Is MSan superior over Memcheck?

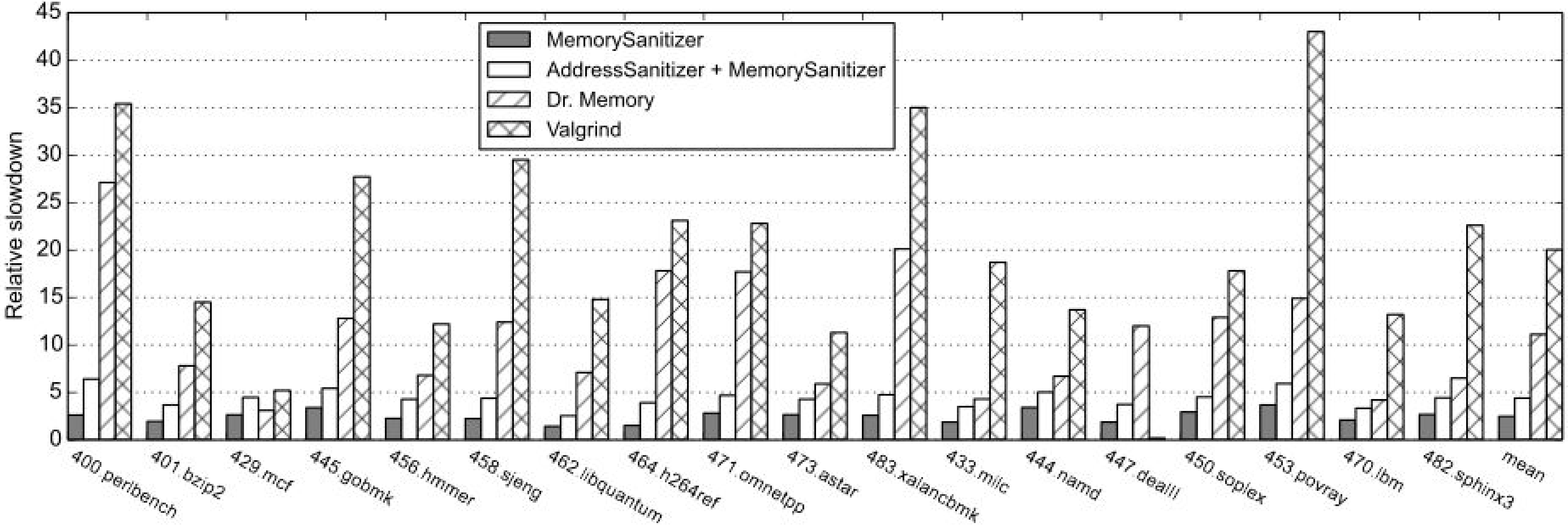
Used for: **Chrome**, **LLVM**

Example:

proprietary console app, 1.3 MLOC in C++

Memory Sanitizer	**Memcheck**
20+ unique bugs in < **2 hours**	the same bugs in **24+ hours**
better reports for stack memory	

Benchmarks



Performance comparison with state-of-the-art tools (SPEC-2006)

Benchmarks

Benchmark	Base	MSan	MSan/O	Valgrind	Valgrind/O	Dr. Memory
Clang	17	106	118	4525	6053	828
Chromium	586	898	1257	97996	158230	n/a

Application startup time (ms) comparison

MemorySanitizer - features

is bit-exact

able to track origins

is significantly faster than Memcheck

causes **3x** slowdown, uses **2x** more real memory

requires that all program code is instrumented

supports Linux x86_64 only, ASLR has to be turned on

References

MemorySanitizer*: *fast detector of uninitialized memory use in C++:
Evgeniy Stepanov (Google), Kostya Serebryany (Google)

http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf

MemorySanitizer:
Evgeniy Stepanov (Google), Kostya Serebryany (Google), 2013

http://llvm.org/devmtg/2013-04/stepanov-slides.pdf

How to Shadow Every Byte of Memory Used by a Program:
Nicholas Nethercote (National ICT Australia), Julian Seward (Open Works LLP), 2007

http://valgrind.org/docs/shadow-memory2007.pdf

AddressSanitizer - ASan

is a detector of **memory errors**:

AddressSanitizer - ASan

is a detector of **memory errors**:

use after free (dangling pointer dereference)

use after return

AddressSanitizer - ASan

is a detector of **memory errors**:

use after free (dangling pointer dereference)

use after return

stack buffer overflow

heap buffer overflow

global buffer overflow

AddressSanitizer - ASan

is a detector of **memory errors**:

use after free (dangling pointer dereference)

use after return

stack buffer overflow

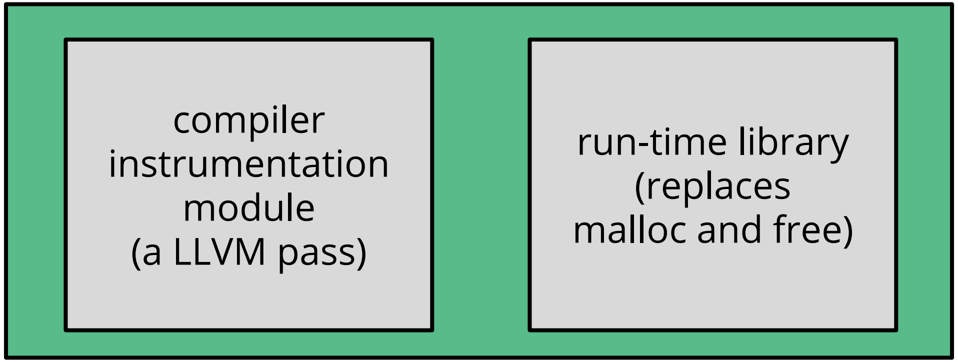
heap buffer overflow

global buffer overflow

initialization order bugs

AddressSanitizer - ASan

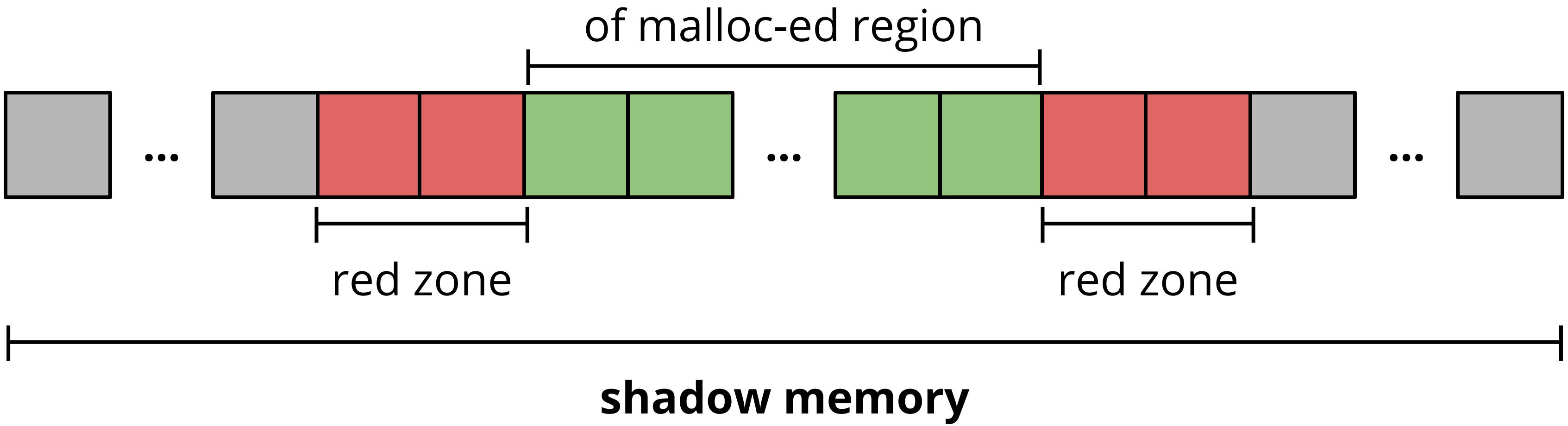
The tool works on x86 Linux and Mac, and ARM Android.




```
%clang -fsanitize=address -fPIE -pie -fno-omit-frame-pointer -g -O1 ex.cc
```

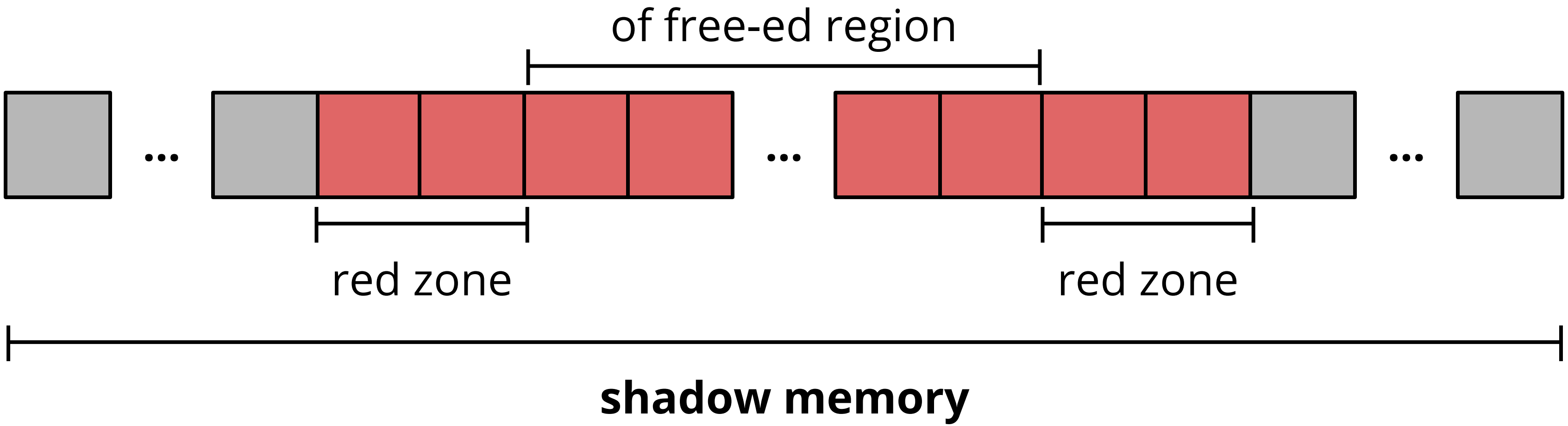
AddressSanitizer - the idea

The memory around malloc-ed regions (**red zones**) is poisoned.



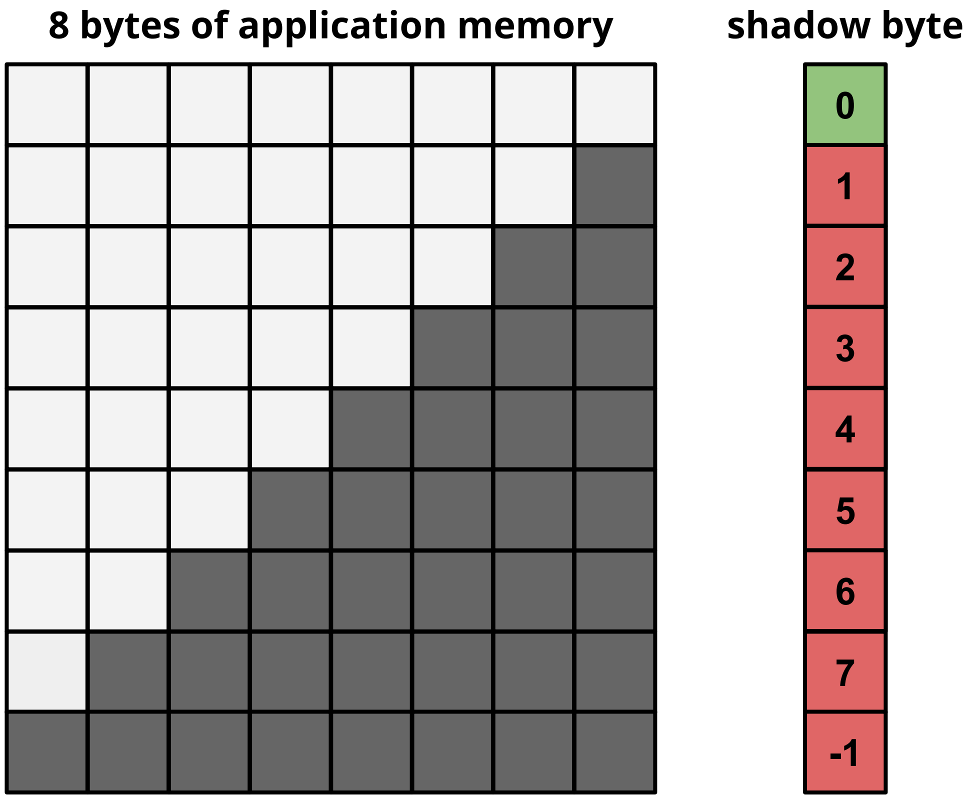
AddressSanitizer - the idea

The free-ed memory is poisoned and put in **quarantine**:
this chunk will not be returned again by malloc in the nearest future.



Shadow memory in ASan

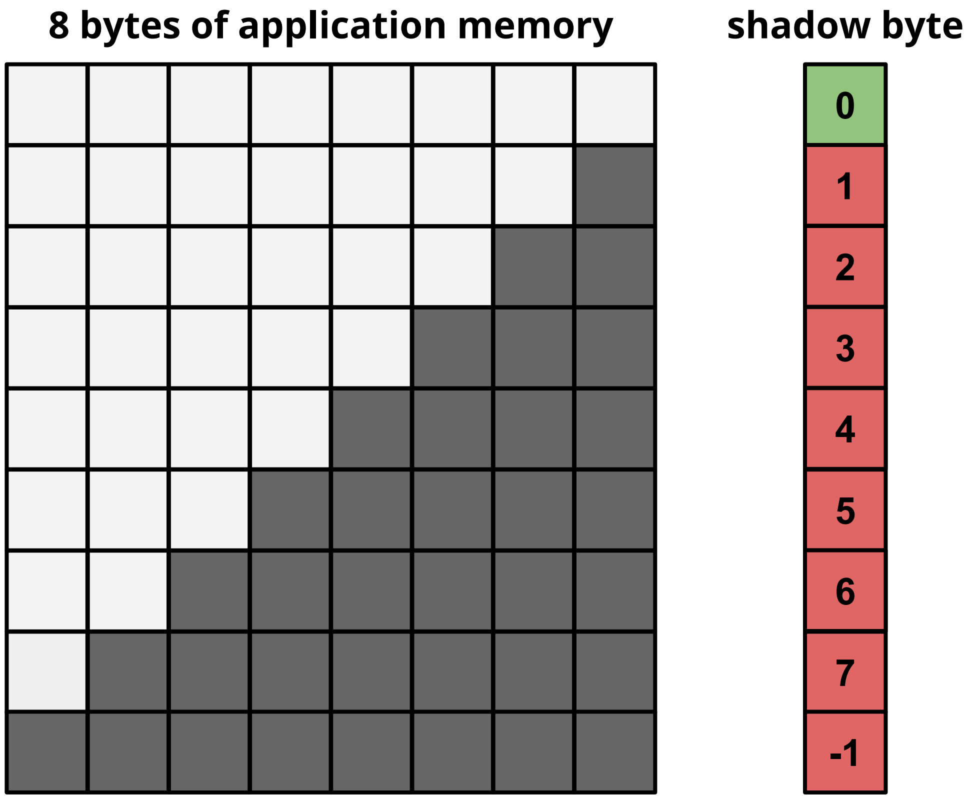
`malloc` returns 8-byte aligned chunks of memory: a **tail** may be addressable only partially.



Shadow memory in ASan

`malloc` returns 8-byte aligned chunks of memory: a **tail** may be addressable only partially.

A chunk's state informs how many of first bytes are addressable.

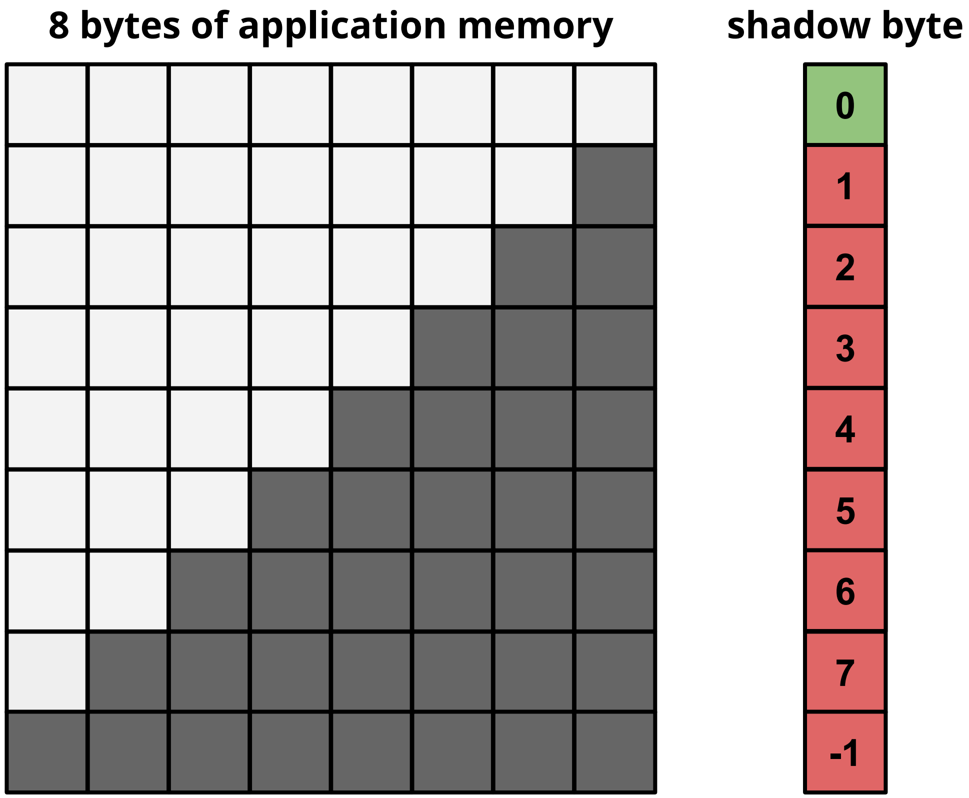


Shadow memory in ASan

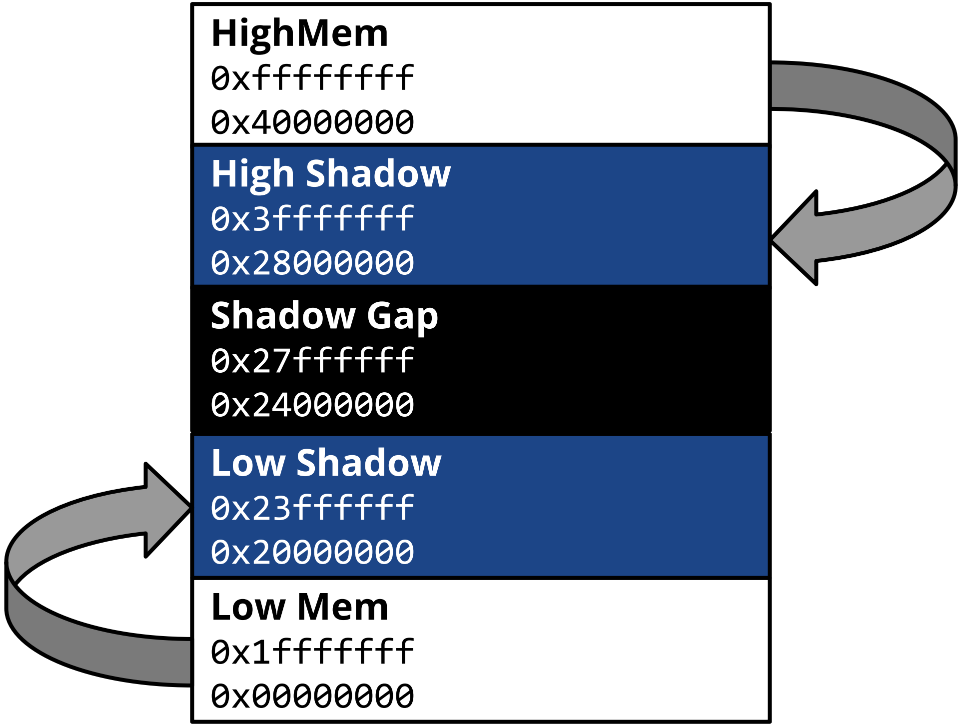
`malloc` returns 8-byte aligned chunks of memory: a **tail** may be addressable only partially.

A chunk's state informs how many of first bytes are addressable.

Every aligned **8-byte word** of memory has only **9 states**.



Shadow mapping in ASan (32 bit)



`shadow = (addr >> 3) + 0x20000000`

AddressSanitizer - the idea

Every memory access in the program is transformed by the compiler:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}

*address = …; // or: … = *address
```

AddressSanitizer - the idea

Every memory access in the program is transformed by the compiler:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}

*address = ...; // or: ... = *address
```

`IsPoisoned` needs to:

access shadow memory for a given address
and verify its state

AddressSanitizer - the idea

Every memory access in the program is transformed by the compiler:

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}

*address = ...; // or: ... = *address
```

`IsPoisoned` needs to:

access shadow memory for a given address
and verify its state

`FAST.`

Instrumentation: IsPoisoned

```
byte *shadow_address = MemToShadow(address);
byte shadow_value = *shadow_address;
if (shadow_value) {
  if (SlowPathCheck(shadow_value, address, kAccessSize)) {
    ReportError(address, kAccessSize, kIsWrite);
  }
}
```

https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

Instrumentation: IsPoisoned

```
byte *shadow_address = MemToShadow(address);
byte shadow_value = *shadow_address;
if (shadow_value) {
  if (SlowPathCheck(shadow_value, address, kAccessSize)) {
    ReportError(address, kAccessSize, kIsWrite);
  }
}
```

MemToShadow for shadow memory returns unaddressable **shadow gap**.

https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

Instrumentation: IsPoisoned

```
byte *shadow_address = MemToShadow(address);
byte shadow_value = *shadow_address;
if (shadow_value) {
  if (SlowPathCheck(shadow_value, address, kAccessSize)) {
    ReportError(address, kAccessSize, kIsWrite);
  }
}

// Check the cases where we access first k bytes of the qword
// and these k bytes are unpoisoned.
bool SlowPathCheck(shadow_value, address, kAccessSize) {
  last_accessed_byte = (address & 7) + kAccessSize - 1;
  return (last_accessed_byte >= shadow_value);
}
```

MemToShadow for shadow memory returns unaddressable **shadow gap**.

https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

How to detect stack buffer overflow?

```
void foo() {
  char a[8];
  ...
  return;
}
```

```
void foo() {
  char redzone1[32];  // 32-byte aligned
  char a[8];
  char redzone2[24];  // 32-byte aligned
  char redzone3[32];  // 32-byte aligned
  int  *shadow_base = MemToShadow(redzone1);
  shadow_base[0] = 0xffffffff;  // poison redzone1
  shadow_base[1] = 0xffffff00;  // poison redzone2, unpoison 'a'
  shadow_base[2] = 0xffffffff;  // poison redzone3
  ...
  // unpoison all
  shadow_base[0] = shadow_base[1] = shadow_base[2] = 0;
  return;
}
```

When AddressSanitizer finds a bug:

it calls one of the functions:

`__asan_report_{load, store}``{1, 2, 4, 8, 16}`

↓

`__asan_report_error`

When AddressSanitizer finds a bug:

it calls one of the functions:

`__asan_report_{load, store}``{1, 2, 4, 8, 16}`

↓

`__asan_report_error`

So if you want gdb to stop **before** ASan report an error:

set breakpoint on `__asan_report_error`

ASan with gdb

To stop gdb **before** ASan report an error:

set breakpoint on `__asan_report_error`

https://github.com/google/sanitizers/wiki/AddressSanitizerAndDebugger

ASan with gdb

To stop gdb **before** ASan report an error:

set breakpoint on `__asan_report_error`

To stop gdb **after** ASan has reported an error:

set breakpoint on `AsanDie`

https://github.com/google/sanitizers/wiki/AddressSanitizerAndDebugger

ASan with gdb

To stop gdb **before** ASan report an error:

set breakpoint on `__asan_report_error`

To stop gdb **after** ASan has reported an error:

set breakpoint on `AsanDie`

To see information on a memory location:

```
(gdb) set overload-resolution off
(gdb) p __asan_describe_address(0x7ffff73c3f80)
```

https://github.com/google/sanitizers/wiki/AddressSanitizerAndDebugger

Good news

There is **no need to recompile shared libraries**:

ASan will work even if you rebuild just part of your program!

Good news

There is **no need to recompile shared libraries**:

ASan will work even if you rebuild just part of your program!

AddressSanitizer is **not expected to produce false positives**:

if you see one, look again - most likely it is a true positive!

Good news

There is **no need to recompile shared libraries**:

ASan will work even if you rebuild just part of your program!

AddressSanitizer is **not expected to produce false positives**:

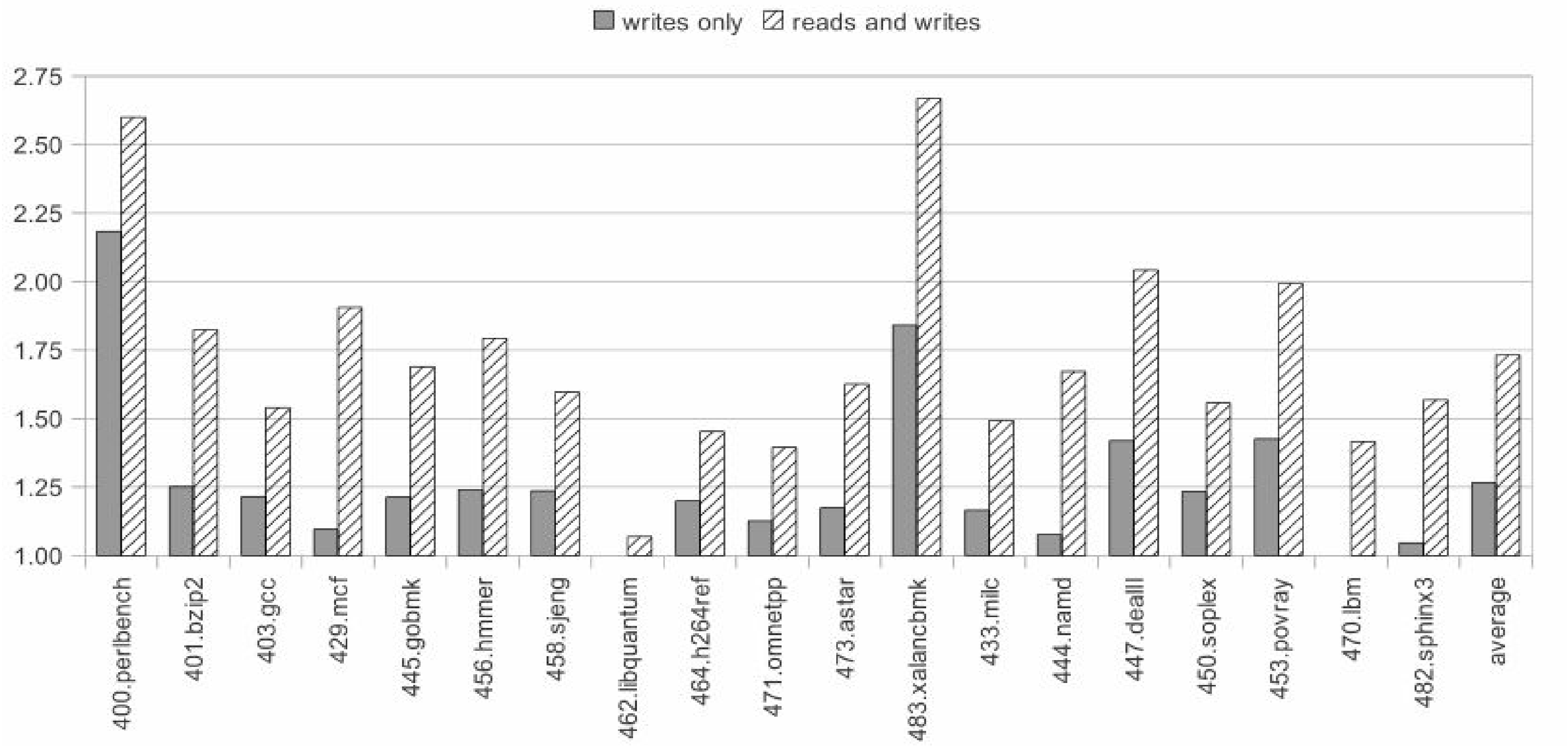
if you see one, look again - most likely it is a true positive!

Real-life **performance** is great:

Almost no slowdown for GUI programs!

Typical overall **memory overhead is 2x - 4x**!

Benchmarks



The average slowdown on 64-bit Linux (SPEC-2006)

Difficulties

There are **very few optimizations** implemented.

Because of specific dataflow some address checks are redundant.

Difficulties

There are **very few optimizations** implemented.

Because of specific dataflow some address checks are redundant.

Some **false negatives** can occur.

With compacted mapping ASan does not catch unaligned partially out-of-bound accesses:

Difficulties

There are **very few optimizations** implemented.

Because of specific dataflow some address checks are redundant.

Some **false negatives** can occur.

With compacted mapping ASan does not catch unaligned partially out-of-bound accesses:

```
int *x = new int[2];            // 8 bytes: [0,7]
int *u = (int*)((char*)x + 6);
*u = 1;                         // Access to range [6-9]
```

Is ASan superior over Memcheck?

	Valgrind	AddressSanitizer
Heap out-of-bounds	**YES**	**YES**
Stack out-of-bounds	**NO**	**YES**
Global out-of-bounds	**NO**	**YES**
Use-after-free	**YES**	**YES**
Use-after-return	**NO**	**Sometimes/YES**
Uninitialized reads	**YES**	**NO**
Overhead	**10x-30x**	**1.5x-3x**

ASan in use

The tool has been applied **Chromium** with **WebKit**.

In first 10 months of using ASan detected:

heap-use-after-free	**201**
heap-buffer-overflow	**73**
global-buffer-overflow	**8**
stack-buffer-overflow	**7**

References

Finding races and memory errors with LLVM instrumentation:
Timur Iskhodzhanov, Alexander Potapenko, Alexey Samsonov, **Kostya Serebryany**, Evgeniy Stepanov, Dmitriy Vyukov (Google), 2011

http://llvm.org/devmtg/2011-11/Serebryany_FindingRacesMemoryErrors.pdf

AddressSanitizer: A Fast Address Sanity Checker:
Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov (Google), 2012

http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37752.pdf

Finding races and memory errors with LLVM instrumentation:
Konstantin Serebryany (Google), 2011

https://www.youtube.com/watch?v=CPnRS1nv3_s

LLVM tools

Why LLVM?

Sanitizers

Work in progress

Goals of SAFECode project

How to enforce memory safety?

Goals of SAFECode project

How to enforce memory safety?

Traditional approach:

runtime checks and garbage collection

Goals of SAFECode project

How to enforce memory safety?

Traditional approach:

runtime checks and garbage collection

The SAFEcode approach:

100% static enforcement of memory safety

~~programmer annotations~~, ~~runtime checks~~, ~~garbage collection~~

supposed to work for a large subclass of type-safe C programs

SAFECode compiler

Built using the LLVM Compiler Infrastructure and the Clang compiler driver.

SAFECode is implemented as a set of **LLVM analysis and transform passes.**

SAFECode compiler

Built using the LLVM Compiler Infrastructure and the Clang compiler driver.

SAFECode is implemented as a set of **LLVM analysis and transform passes.**

A **memory safety compiler**: inserts runtime checks into a program to catch memory safety errors at runtime.

SAFECode compiler

Built using the LLVM Compiler Infrastructure and the Clang compiler driver.

SAFECode is implemented as a set of **LLVM analysis and transform passes.**

A **memory safety compiler**: inserts runtime checks into a program to catch memory safety errors at runtime.

With additional instrumentation it can track debugging information.

SAFECode compiler

Built using the LLVM Compiler Infrastructure and the Clang compiler driver.

SAFECode is implemented as a set of **LLVM analysis and transform passes.**

A **memory safety compiler**: inserts runtime checks into a program to catch memory safety errors at runtime.

With additional instrumentation it can track debugging information.

Work in progress...?

ThreadSanitizer (tsan)

is a data race detector for C/C++ programs

ThreadSanitizer (tsan)

is a data race detector for C/C++ programs

Linux and Mac versions are based on Valgrind framework

ThreadSanitizer (tsan)

is a data race detector for C/C++ programs

Linux and Mac versions are based on Valgrind framework

What makes tsan better than Helgrind?

It provides a hybrid mode, which may give more false positives, but is **much faster**, **more predictable** and **find more real races**.

https://code.google.com/p/data-race-test/wiki/ThreadSanitizerVsOthers

ThreadSanitizerVsOthers

Comparison of ThreadSanitizer, Helgrind, Drd and Intel Thread Checker

Updated Feb 4, 2010 by konstant...@gmail.com

UNDER CONSTRUCTION!

Some features that differ ThreadSanitizer from Helgrind (and also from DRD and Intel Thread Checker).

ThreadSanitizer has both **hybrid** and **pure happens-before** state machines while such detectors as Helgrind (3.4), DRD, and Intel Thread Checker use only pure happens-before machine.

> The pure happens-before mode will not report false positives (unless your program uses lock-less synchronization), but it may miss races and is less predictable.
>
> The hybrid machine may give more false positives, but is much faster, more predictable and find more real races.

ThreadSanitizer supports DynamicAnnotations which can make any tricky synchronization (including lock-less synchronization) to be ThreadSanitizer-friendly.

ThreadSanitizer prints all accesses involved in a data race and also all locks held during each access. For details see ThreadSanitizerAlgorithm and the screenshot at the main ThreadSanitizer page.

ThreadSanitizer has an *ignore* feature which is complementary to valgrind suppressions.

ThreadSanitizer does not replace the application's **malloc**, but gently instruments it. This is usefull if the application uses a custom malloc function (e.g. Google's TCMalloc) which has important side effects.

ThreadSanitizer is written in C++ with STL. This is, AFAIK, the first valgrind tool written in C++. Not a big deal otherwise. :)

https://code.google.com/p/data-race-test/wiki/ThreadSanitizerVsOthers

ThreadSanitizerVsOthers

Comparison of ThreadSanitizer, Helgrind, Drd and Intel Thread Checker

Updated Feb 4, 2010 by konstant...@gmail.com

UNDER CONSTRUCTION!

Some features that differ ThreadSanitizer from Helgrind (and also from DRD and Intel Thread Checker).

ThreadSanitizer has both **hybrid** and **pure happens-before** state machines while such detectors as Helgrind (3.4), DRD, and Intel Thread Checker use only pure happens-before machine.

> The pure happens-before mode will not report false positives (unless your program uses lock-less synchronization), but it may miss races and is less predictable.
>
> The hybrid machine may give more false positives, but is much faster, more predictable and find more real races.

ThreadSanitizer supports DynamicAnnotations which can make any tricky synchronization (including lock-less synchronization) to be ThreadSanitizer-friendly.

ThreadSanitizer prints all accesses involved in a data race and also all locks held during each access. For details see ThreadSanitizerAlgorithm and the screenshot at the main ThreadSanitizer page.

ThreadSanitizer has an *ignore* feature which is complementary to valgrind suppressions.

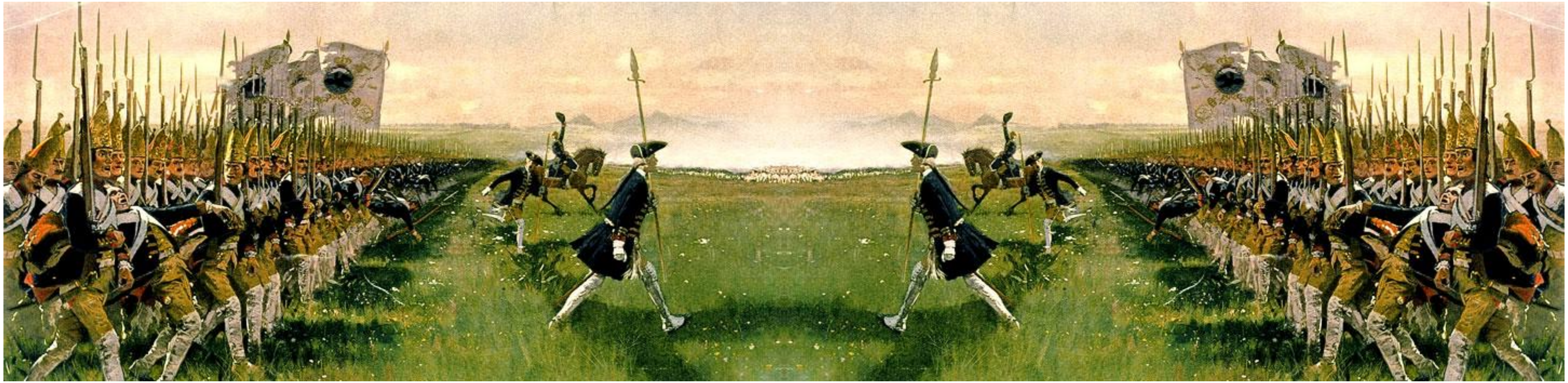
ThreadSanitizer does not replace the application's **malloc**, but gently instruments it. This is usefull if the application uses a custom malloc function (e.g. Google's TCMalloc) which has important side effects.

ThreadSanitizer is written in C++ with STL. This is, AFAIK, the first valgrind tool written in C++. Not a big deal otherwise. :)

"Eternal War in Memory"

defensive research

offensive research



new protections

new attacks

"Eternal War in Memory"

defensive research

offensive research

new protections

new attacks

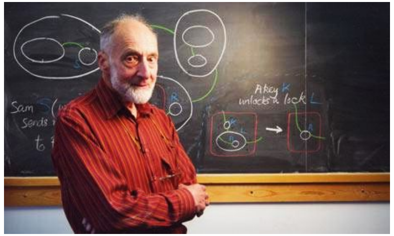
Mobile System Software Group

Inga Rüb (inga.roksana.rueb@gmail.com)

Pictures:

https://upload.wikimedia.org/wikipedia/commons/9/9a/Hohenfriedeberg_-_Attack_of_Prussian_Infantry_-_1745.jpg

http://www.academia.dk/Blog/wp-content/uploads/CanaryInACoalMine_2.jpg



(www.theguardian.com)

http://static.guim.co.uk/sys-images/Guardian/Pix/pictures/2010/4/1/1270144400739/Robin-Milner-001.jpg

http://llvm.org/img/DragonFull.png