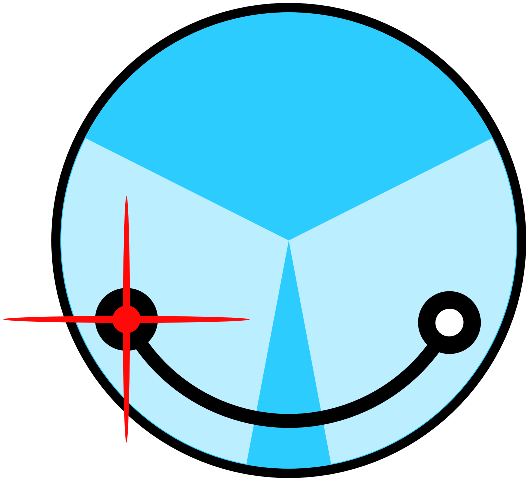


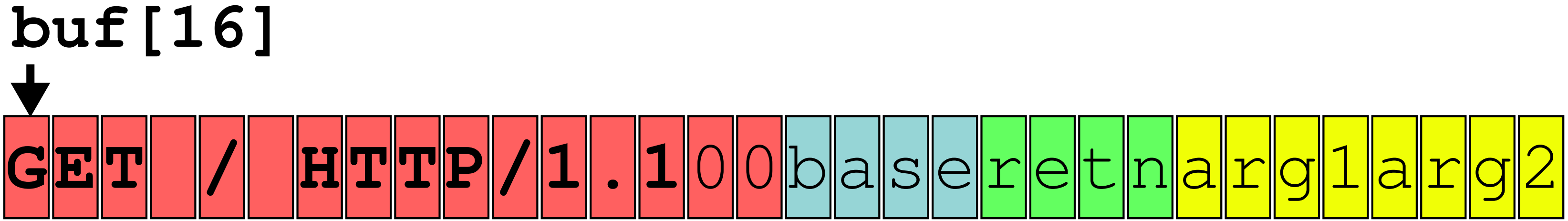
Practical taint analysis for protecting buggy binaries

So your exploit beats ASLR/DEP? I don't care

Erik Bosman <erik@minemu.org>

VRIJE UNIVERSITEIT AMSTERDAM

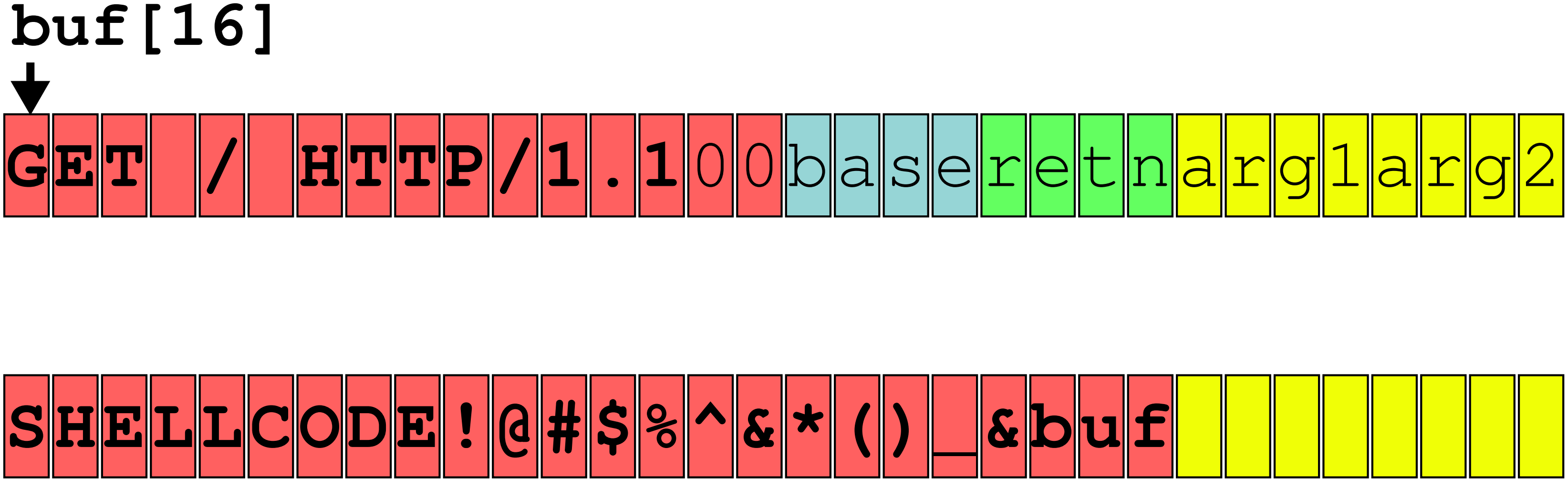
Traditional Stack Smashing



buf[16]

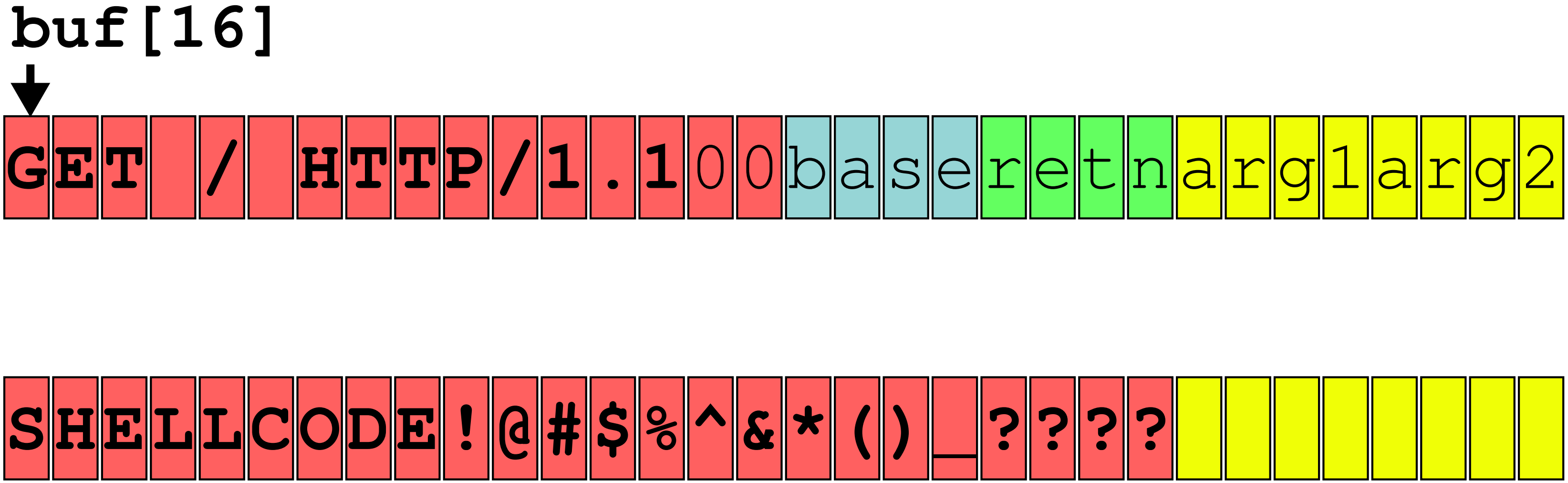
G	E	T		/		H	T	T	P	/	1	.	1	0	0	b	a	s	e	r	e	t	n	a	r	g	1	a	r	g	2

Traditional Stack Smashing



```
buf[16]
GET / HTTP/1.100baseretnarg1arg2
SHELLCODE!@#$%^&*()_&buf
```

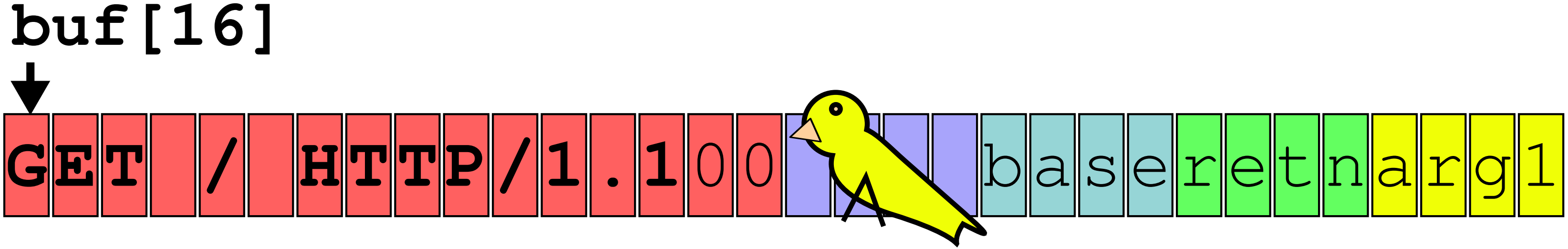
Address Space Layout Randomisation (ASLR)



buf[16]

```
GET / HTTP/1.100baseretnarg1arg2

SHELLCODE!@#$%^&*()_????
```

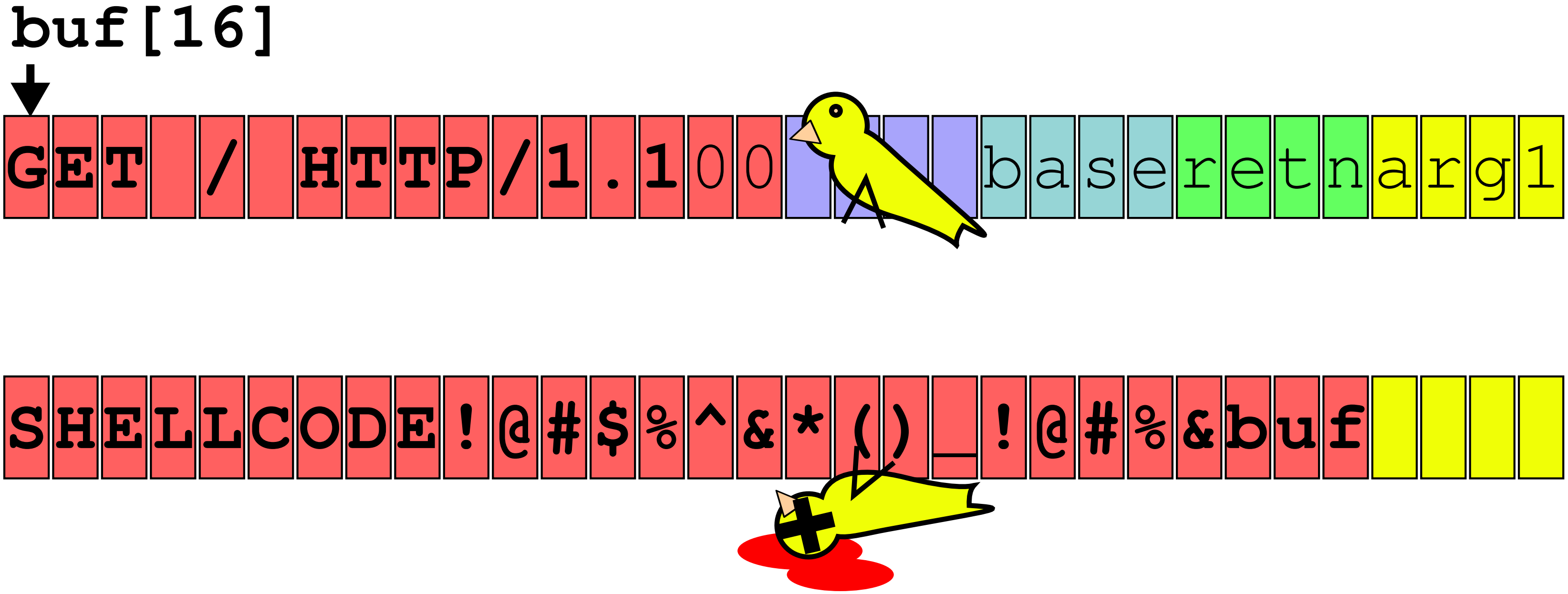
Stack Canaries



buf[16]

G	E	T		/		H	T	T	P	/	1	.	1	0	0					b	a	s	e	r	e	t	n	a	r	g	1

Stack Canaries

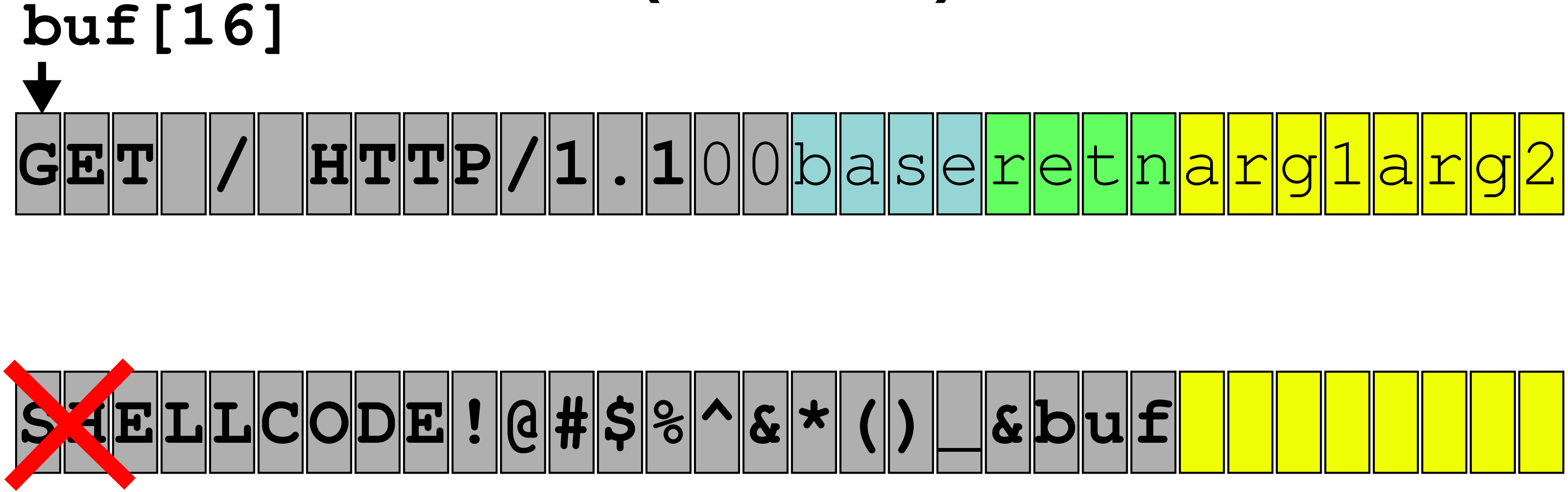


buf[16]

GET / HTTP/1.100   baseretnarg1

SHELLCODE!@#$%^&*()_!@#%&buf

Non-executable data
(DEP / NX)

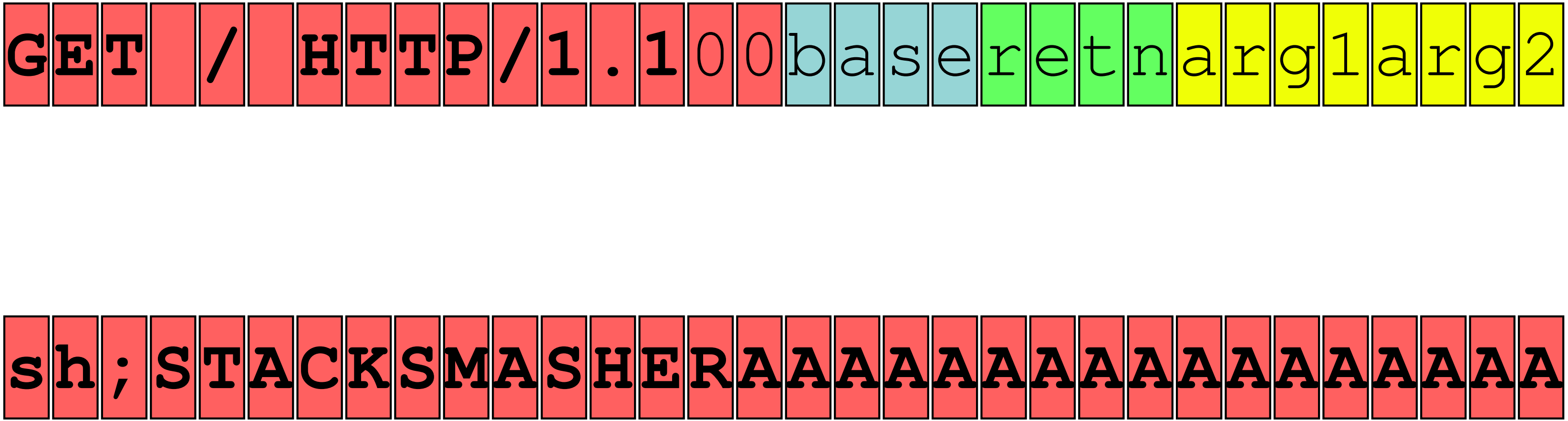


buf[16]

```
GET / HTTP/1.100baseretnarg1arg2

SHELLCODE!@#$%^&*()_&buf
```

Fortify Source

```
char buf[16];
memcpy(buf, r->buf, r->len);
```

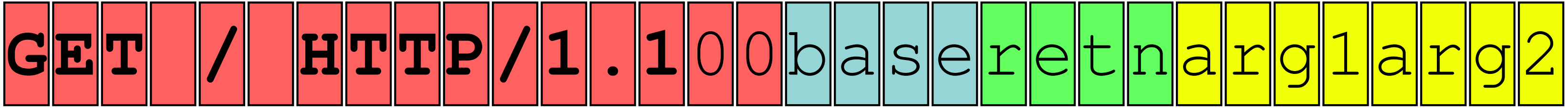


```
GET / HTTP/1.100baseretnarg1arg2
```

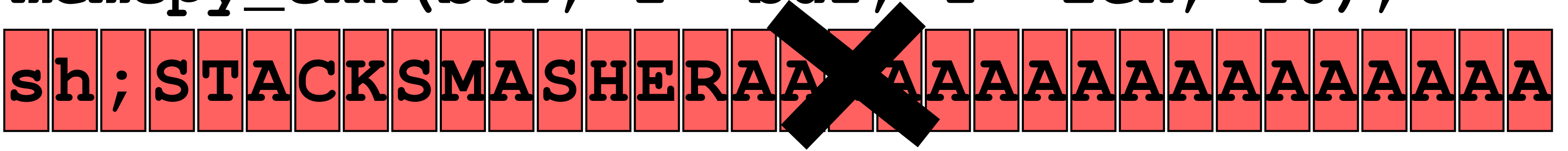
```
sh;STACKSMASHERAAAAAAAAAAAAAAAAA
```

Fortify Source

```
char buf[16];
memcpy(buf, r->buf, r->len);
```



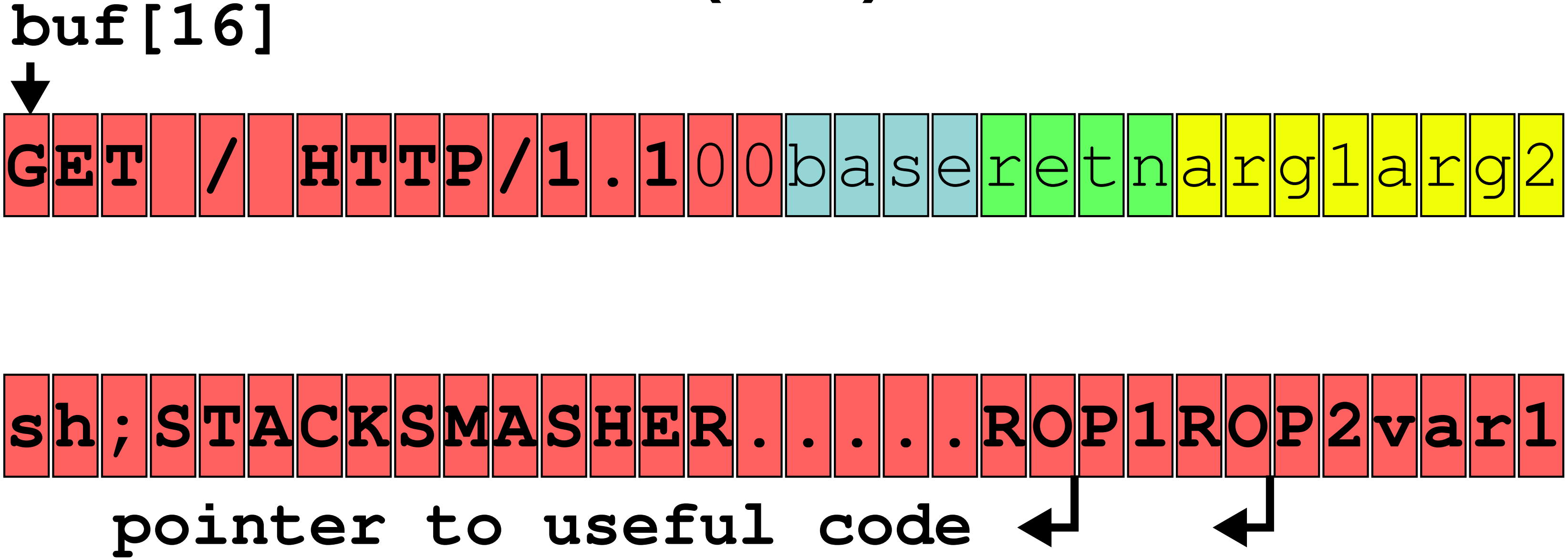
```
char buf[16];
memcpy_chk(buf, r->buf, r->len, 16);
```



```
*** buffer overflow detected ***: /my/fortified/binary terminated
======= Backtrace: =========
/lib/i386-linux-gnu/i686/cmov/libc.so.6(__fortify_fail+0x50)[0xb774a4d0]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(+0xe040a)[0xb774940a]
/my/fortified/binary[0x8048458]
/lib/i386-linux-gnu/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0xb767fe46]
/my/fortified/binary[0x8048371]
======= Memory map: ========
08048000-08049000 r-xp 00000000 fe:00 282465     /my/fortified/binary
08049000-0804a000 rw-p 00000000 fe:00 282465     /my/fortified/binary
08600000-08621000 rw-p 00000000 00:00 0          [heap]
b764b000-b7667000 r-xp 00000000 fe:00 131602     /lib/i386-linux-gnu/libgcc_s.so.1
b7667000-b7668000 rw-p 0001b000 fe:00 131602     /lib/i386-linux-gnu/libgcc_s.so.1
b7668000-b7669000 rw-p 00000000 00:00 0
...

Aborted
```

FORTIFY ALL THE THINGS

Return Oriented Programming (ROP)

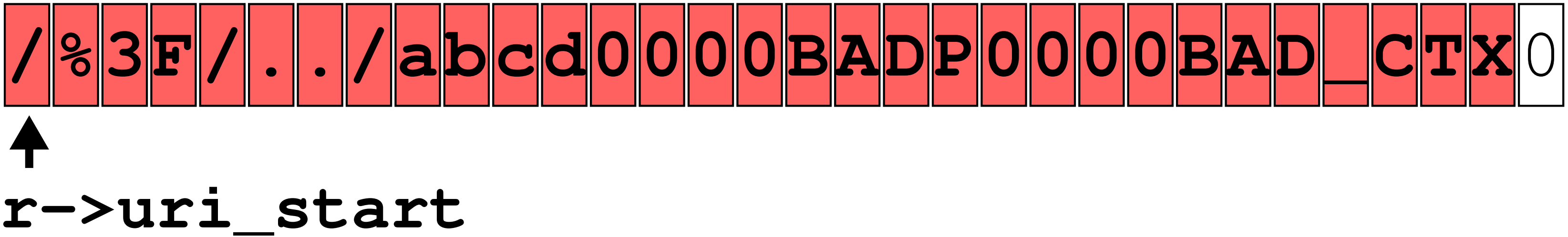


```
buf[16]
GET / HTTP/1.100baseretnarg1arg2

sh;STACKSMASHER.....ROP1ROP2var1
```

Some exploits still work with all these defense measures.

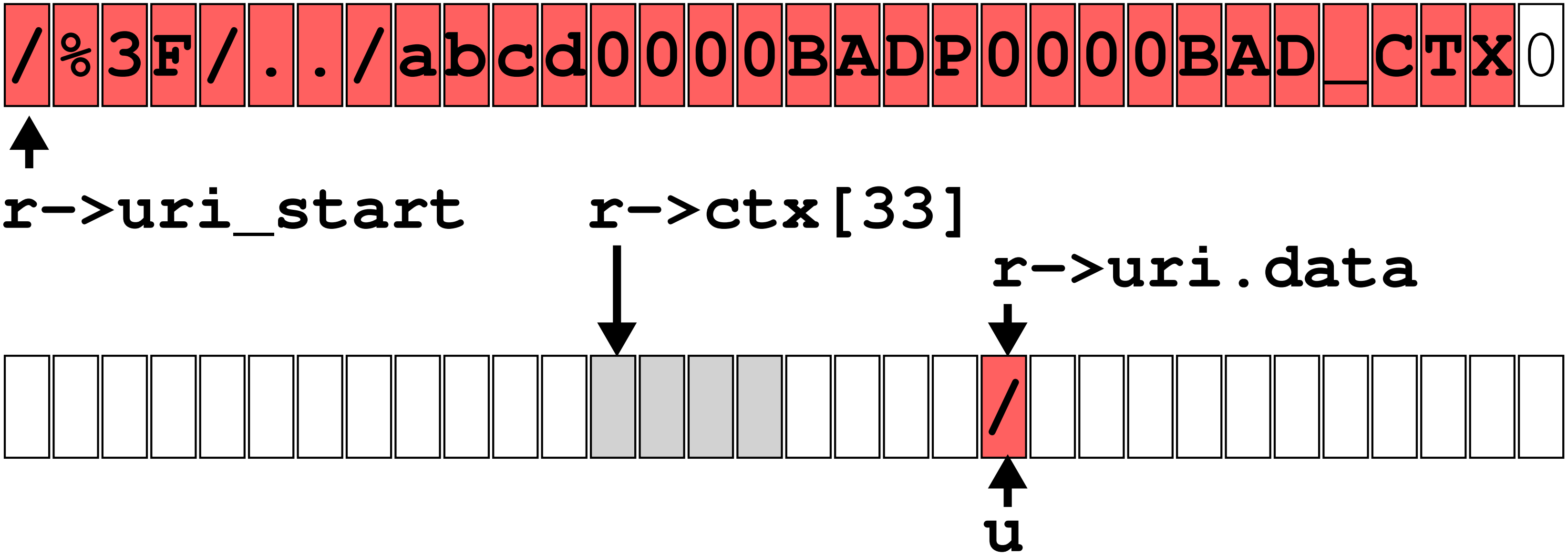
Example: nginx buffer underrun (CVE-2009-2629)

CVE-2009-2629



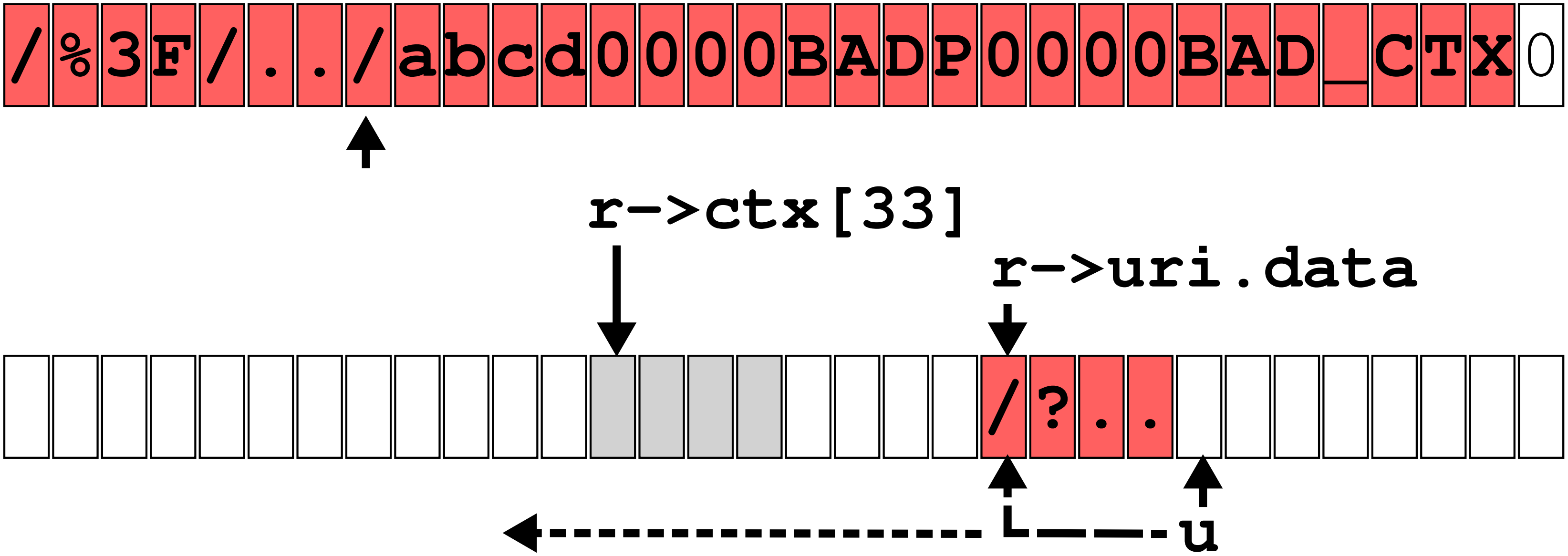
/	%	3	F	/	.	.	/	a	b	c	d	0	0	0	0	B	A	D	P	0	0	0	0	B	A	D	_	C	T	X	0

CVE-2009-2629

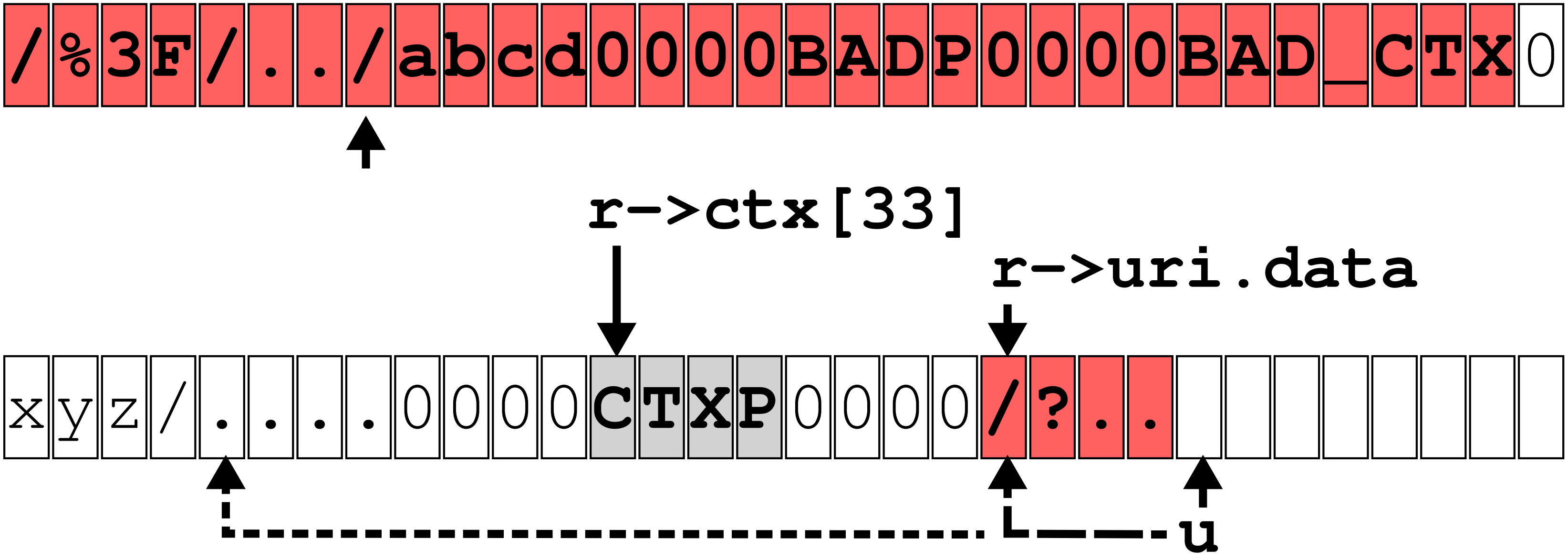


/%3F/../abcd0000BADP0000BAD_CTX0

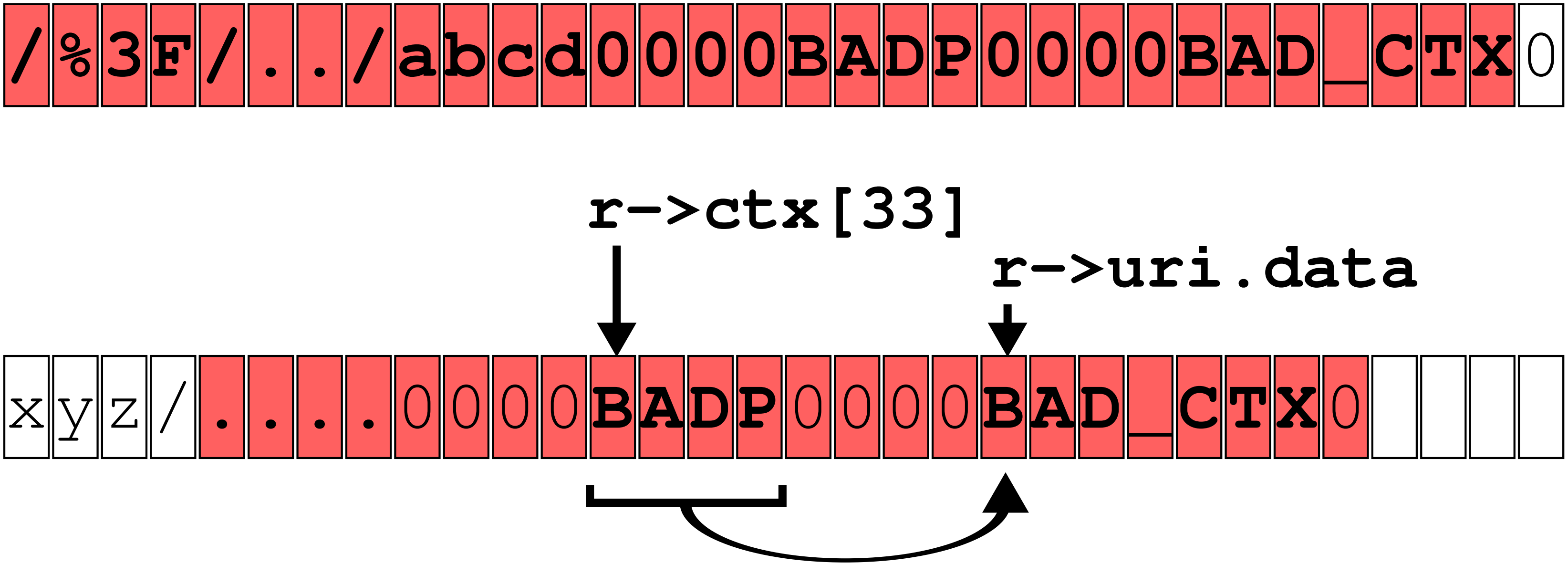
CVE-2009-2629



CVE-2009-2629



CVE-2009-2629



/%3F/../abcd0000BADP0000BAD_CTX0

xyz/....0000BADP0000BAD_CTX0

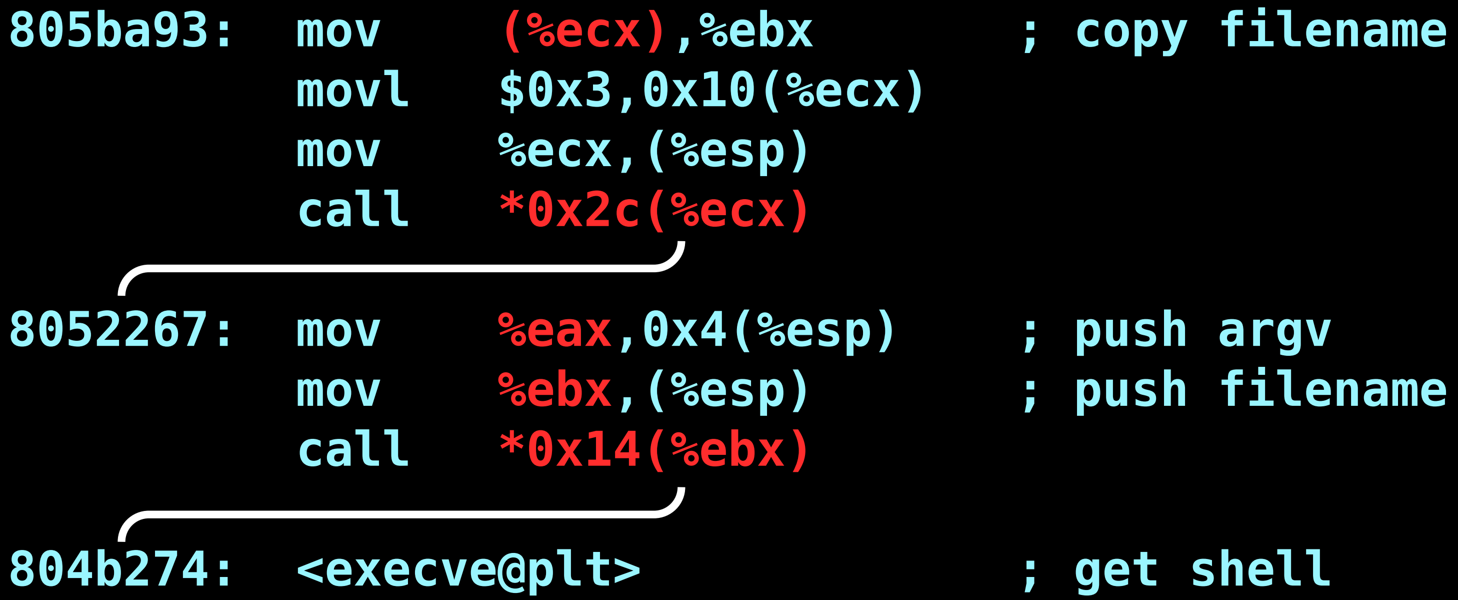
```
typedef struct {
    ngx_buf_t                   *buf;
    ngx_chain_t                 *in;
    ngx_chain_t                 *free;
    ngx_chain_t                 *busy;

    unsigned                     sendfile;
    unsigned                     need_in_memory;
    unsigned                     need_in_temp;

    ngx_pool_t                  *pool;
    ngx_int_t                    allocated;
    ngx_bufs_t                   bufs;
    ngx_buf_tag_t                tag;

    ngx_output_chain_filter_pt   output_filter;
    void                        *filter_ctx;
} ngx_output_chain_ctx_t;
```

```
typedef struct {
    ngx_buf_t                   *buf;
    ngx_chain_t                 *in;
    ngx_chain_t                 *free;
    ngx_chain_t                 *busy;

    unsigned                     sendfile;
    unsigned                     need_in_memory;
    unsigned                     need_in_temp;

    ngx_pool_t                  *pool;
    ngx_int_t                    allocated;
    ngx_bufs_t                   bufs;
    ngx_buf_tag_t                tag;

    ngx_output_chain_filter_pt   output_filter;
    void                        *filter_ctx;
} ngx_output_chain_ctx_t;
```

function pointer

```
805ba93:  mov    (%ecx),%ebx       ; copy filename
          movl   $0x3,0x10(%ecx)
          mov    %ecx,(%esp)
          call   *0x2c(%ecx)
```

```
805ba93:  mov    (%ecx),%ebx          ; copy filename
          movl   $0x3,0x10(%ecx)
          mov    %ecx,(%esp)
          call   *0x2c(%ecx)

8052267:  mov    %eax,0x4(%esp)       ; push argv
          mov    %ebx,(%esp)          ; push filename
          call   *0x14(%ebx)
```



```
805ba93:  mov    (%ecx),%ebx        ; copy filename
          movl   $0x3,0x10(%ecx)
          mov    %ecx,(%esp)
          call   *0x2c(%ecx)

8052267:  mov    %eax,0x4(%esp)     ; push argv
          mov    %ebx,(%esp)        ; push filename
          call   *0x14(%ebx)

804b274:  <execve@plt>              ; get shell
```

- defeats address randomisation (through info leak)

- defeats address randomisation (through info leak)

- defeats non-executable data protection

- defeats address randomisation (through info leak)

- defeats non-executable data protection

- not a standard copy function (no fortify protections)

- defeats address randomisation (through info leak)

- defeats non-executable data protection

- not a standard copy function (no fortify protections)

- not return oriented, so stack smash protection does not matter

But the situation is even worse

But the situation is even worse

- needs to be enabled at compile time, and there is a lot of old code out there

But the situation is even worse

- needs to be enabled at compile time, and there is a lot of old code out there

- many packages do not apply these defence mechanisms even today

But the situation is even worse

- needs to be enabled at compile time, and there is a lot of old code out there

- many packages do not apply these defence mechanisms even today

- implementation flaws

Can we do more?

Can we do more?

>> DEP prevents untrusted data from being run as code

Can we do more?

>> DEP prevents untrusted data from being run as code

<< ROP replaces untrusted code with pointers
to original code.

Can we do more?

>> DEP prevents untrusted data from being run as code

<< ROP replaces untrusted code with pointers
to original code.

>> Can we prevent untrusted pointers from being used
as jump addresses?

Taint analysis

```
0805be60   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   |................|
0805be70   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   |................|
0805be80   00 00 00 00 02 00 00 00  d8 4b 06 08 a0 2e 05 08   |.........K......|
0805be90   94 be 05 08 78 a0 04 08  ef be ad de a4 be 05 08   |....x...........|
0805bea0   ac be 05 08 2f 62 69 6e  2f 73 68 00 a4 be 05 08   |..../bin/sh.....|
0805beb0   00 00 00 00 53 41 4d 45  54 48 49 4e 47 57 45 44   |....SAMETHINGWED|
0805bec0   4f 45 56 45 52 59 4e 49  47 48 54 50 49 4e 4b 59   |OEVERYNIGHTPINKY|
0805bed0   00 00 00 00 4e 41 52 46  90 be 05 08 ef 1f 05 08   |....NARF........|
0805bee0   ff fa 26 08 ff f0 00 00  00 00 00 00 00 00 00 00   |..&.............|
0805bef0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   |................|
0805bf00   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   |................|
```

Taint tracking (1/2):

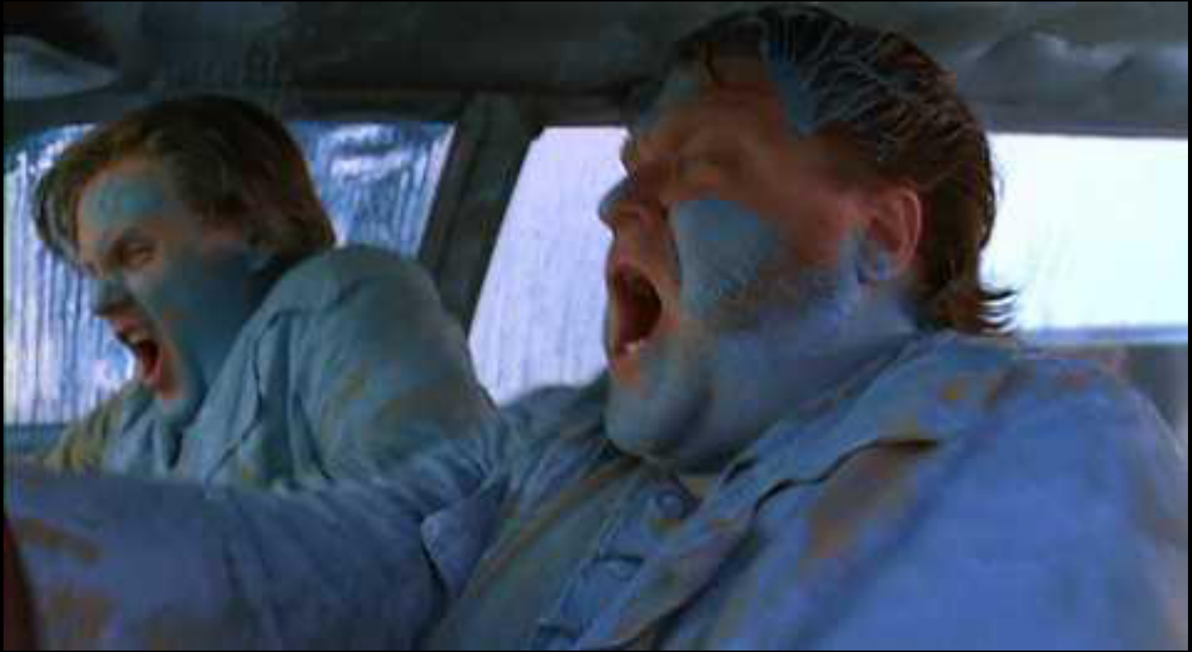
- remember whether data is trusted or not
- untrusted data is 'tainted'
- when data is copied, its taint is copied along
- taint is ORed for arithmetic operations

Taint tracking (2/2):

When the code jumps to an address in memory, the source of this address is checked for taint.

eg.:

- RET
- CALL *%eax
- JMP *0x1c(%ebx)



Taint tracking

photo: sammydavisdog@flickr

useful, but slow as hell

Is this slowness fundamental?

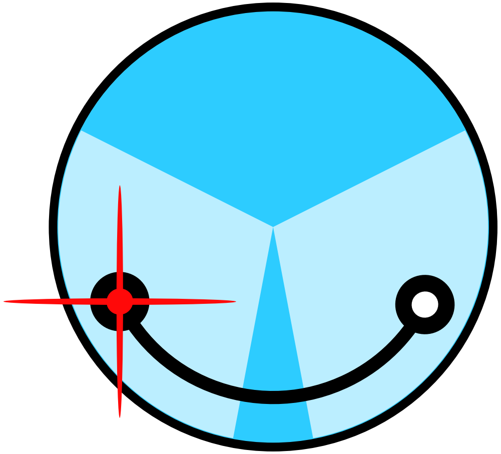
minemu

fast emulator

memory layout

use SSE registers to hold taint

Is this slowness fundamental?



minemu

- fast emulator

memory layout

use SSE registers to hold taint

Emulator

- process-level emulator

Emulator

process-level emulator

- fast x86 -> x86 jit compiler

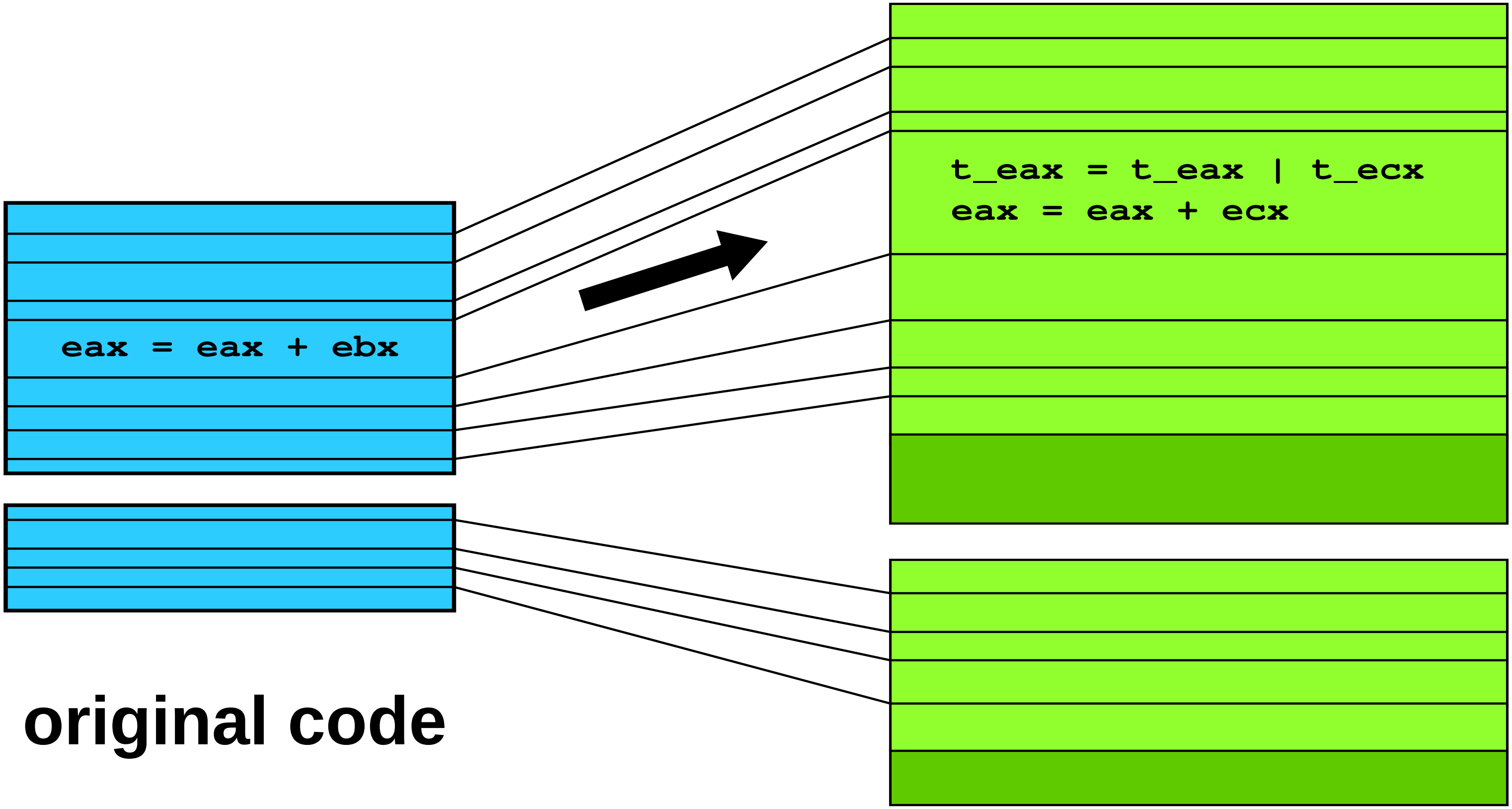
Emulator

process-level emulator

fast x86 -> x86 jit compiler

- keeps register state the same

Emulator



```
eax = eax + ebx
```

```
t_eax = t_eax | t_ecx
eax = eax + ecx
```

original code

jit code

Emulator

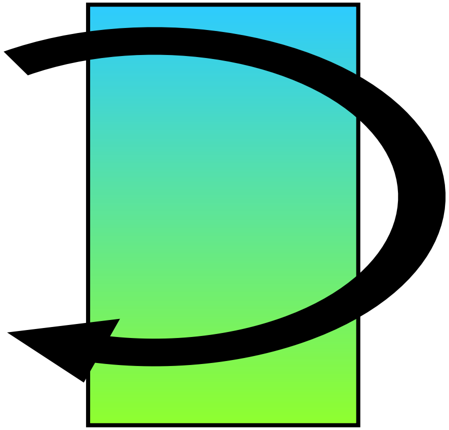
process-level emulator

fast x86 -> x86 jit compiler

keeps register state the same

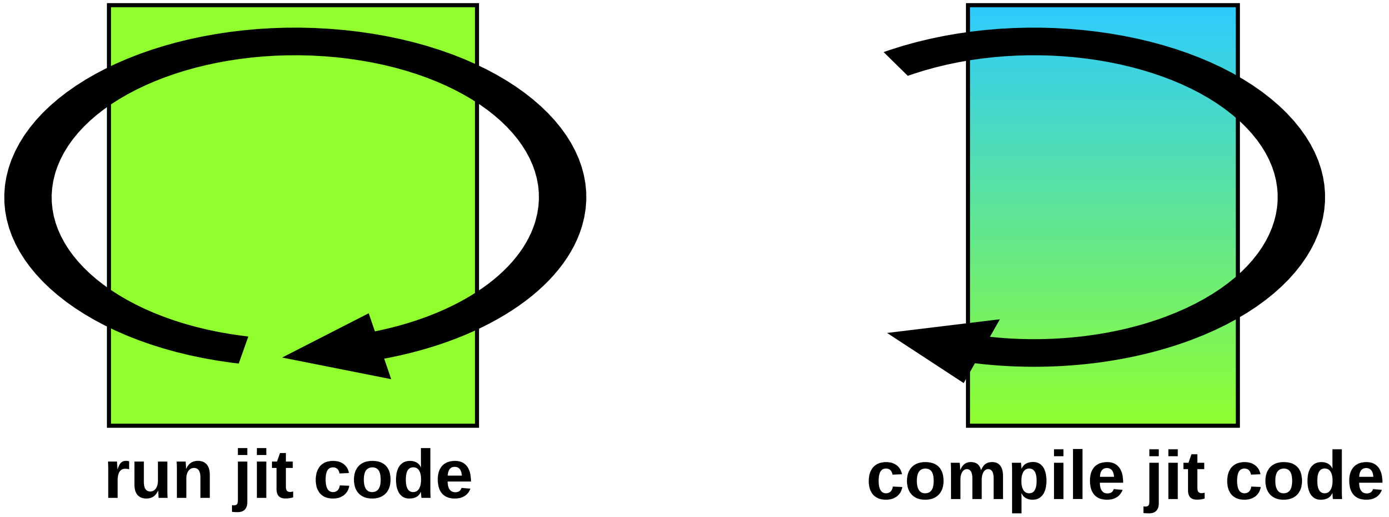
● translates big chunks of code all at once

Emulator



compile jit code

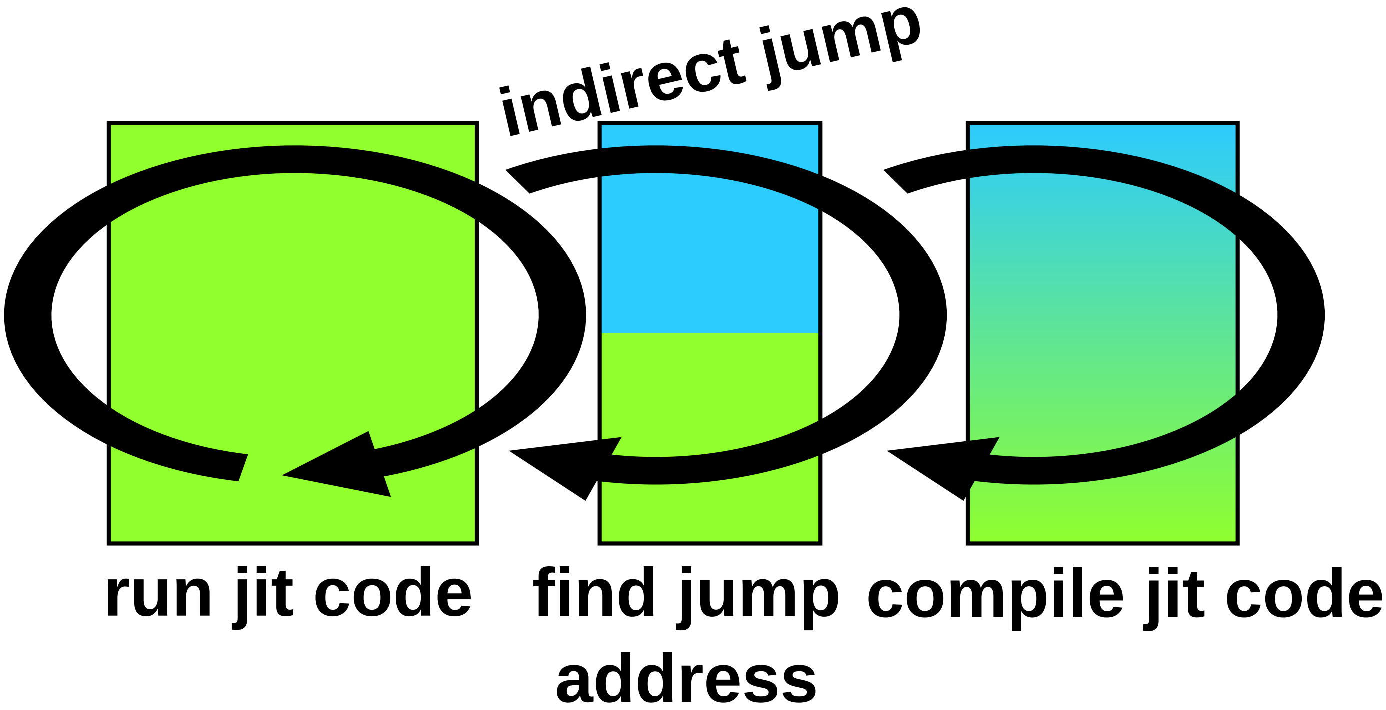
Emulator



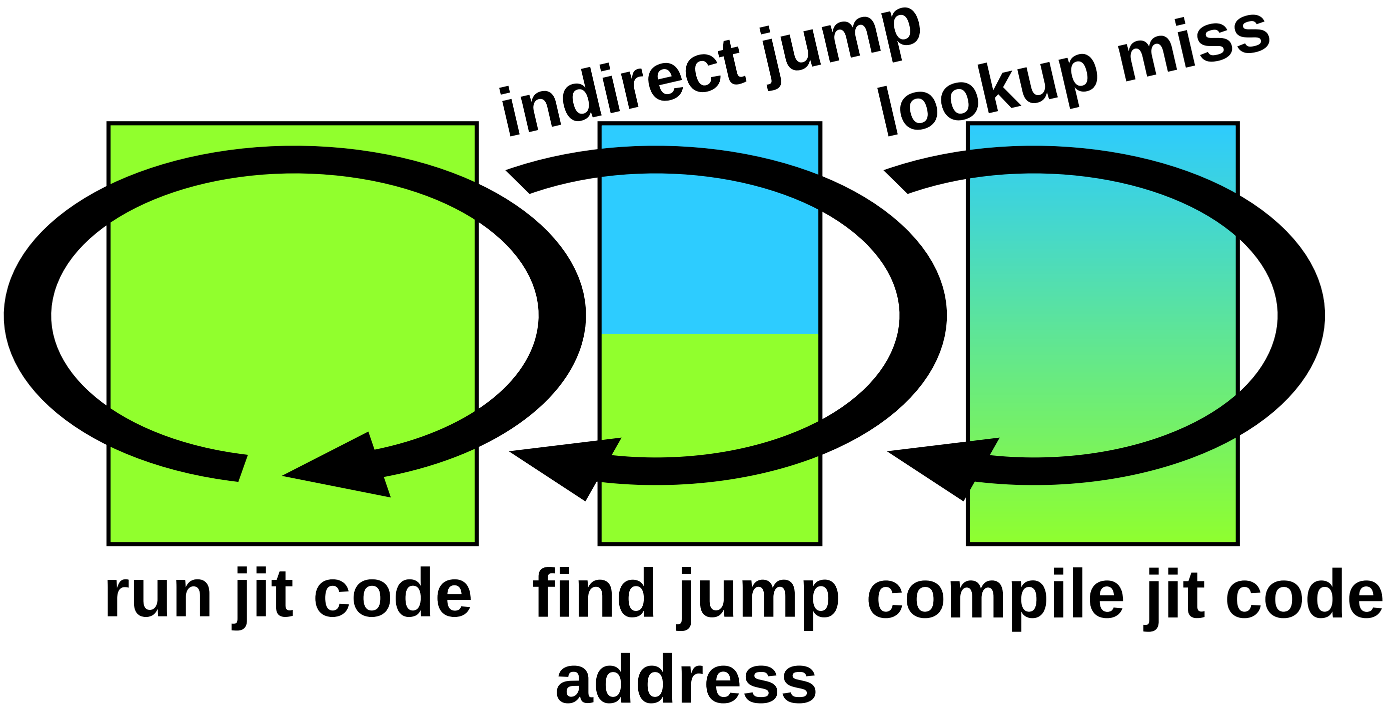
run jit code

compile jit code

Emulator


indirect jump
run jit code
find jump address
compile jit code

Emulator


indirect jump
lookup miss
run jit code
find jump address
compile jit code

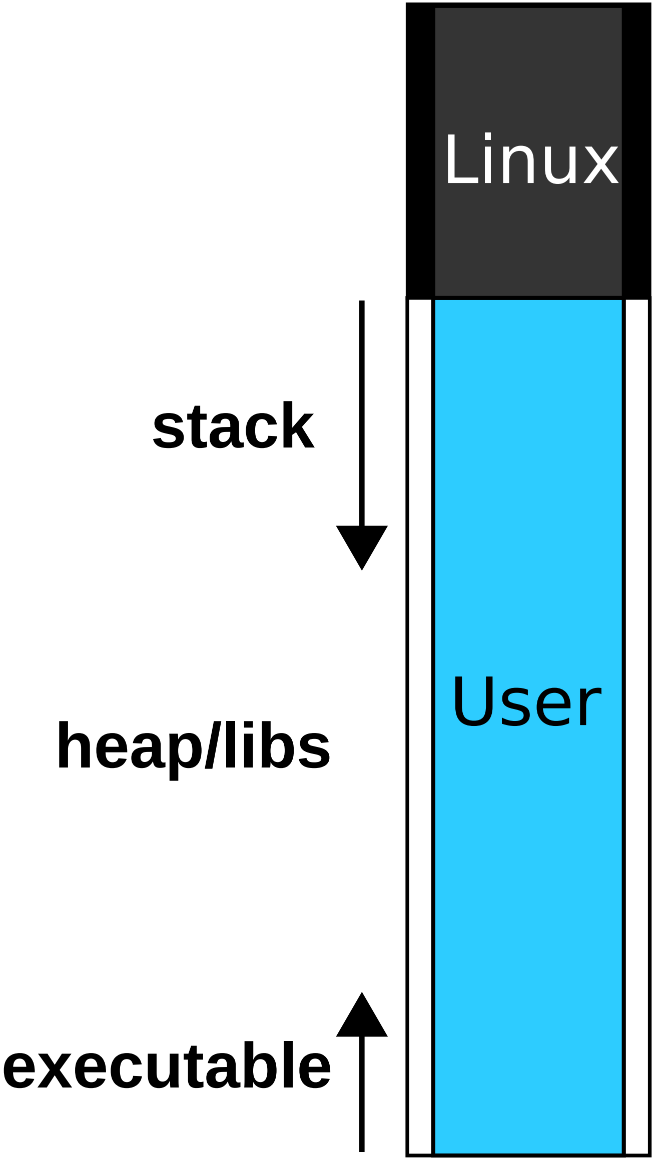
Is this slowness fundamental?

minemu

fast emulator

▶ memory layout

use SSE registers to hold taint



Linux

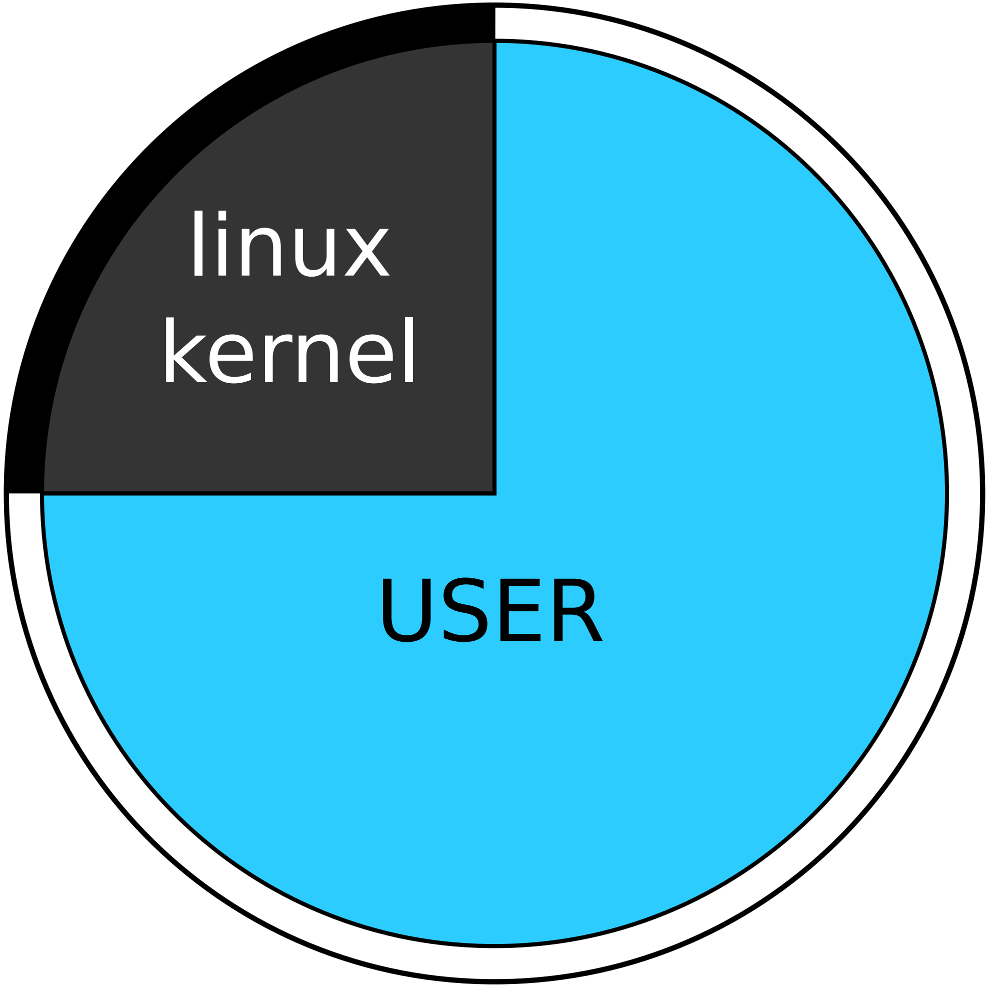
stack

User

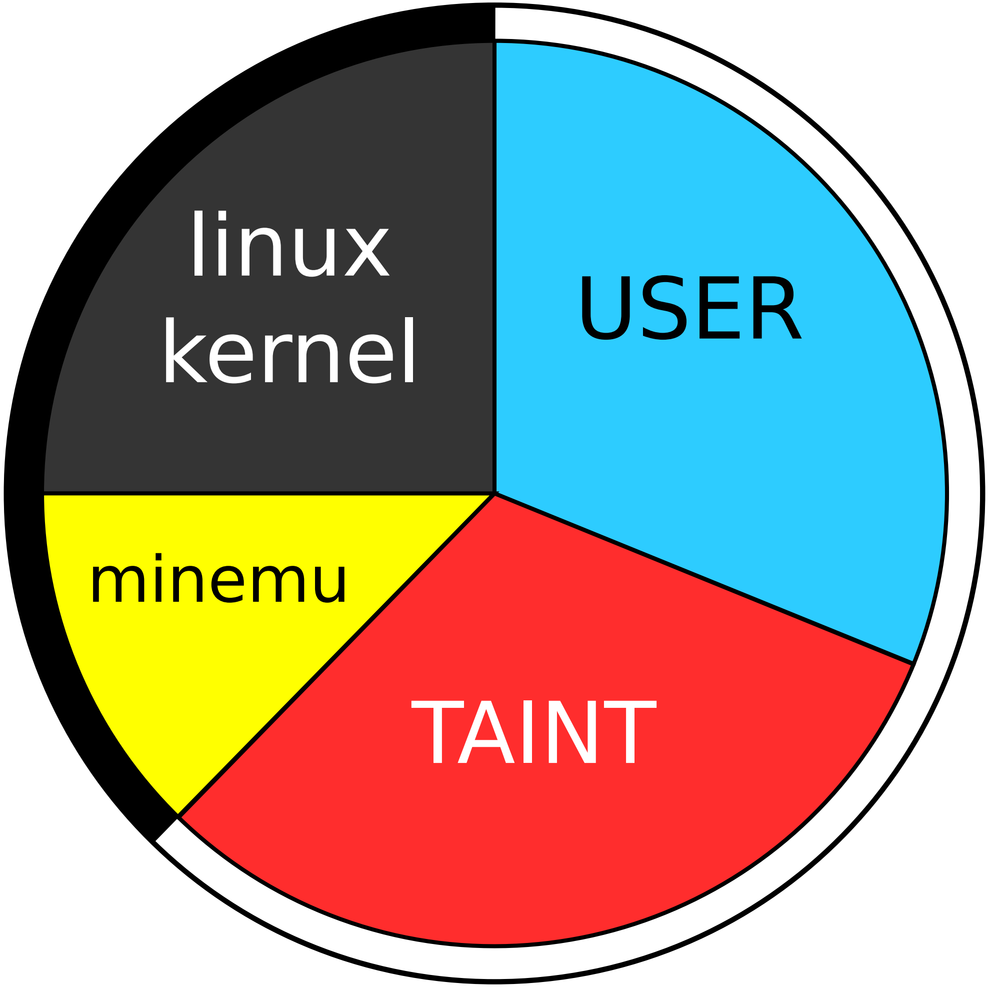
heap/libs

executable

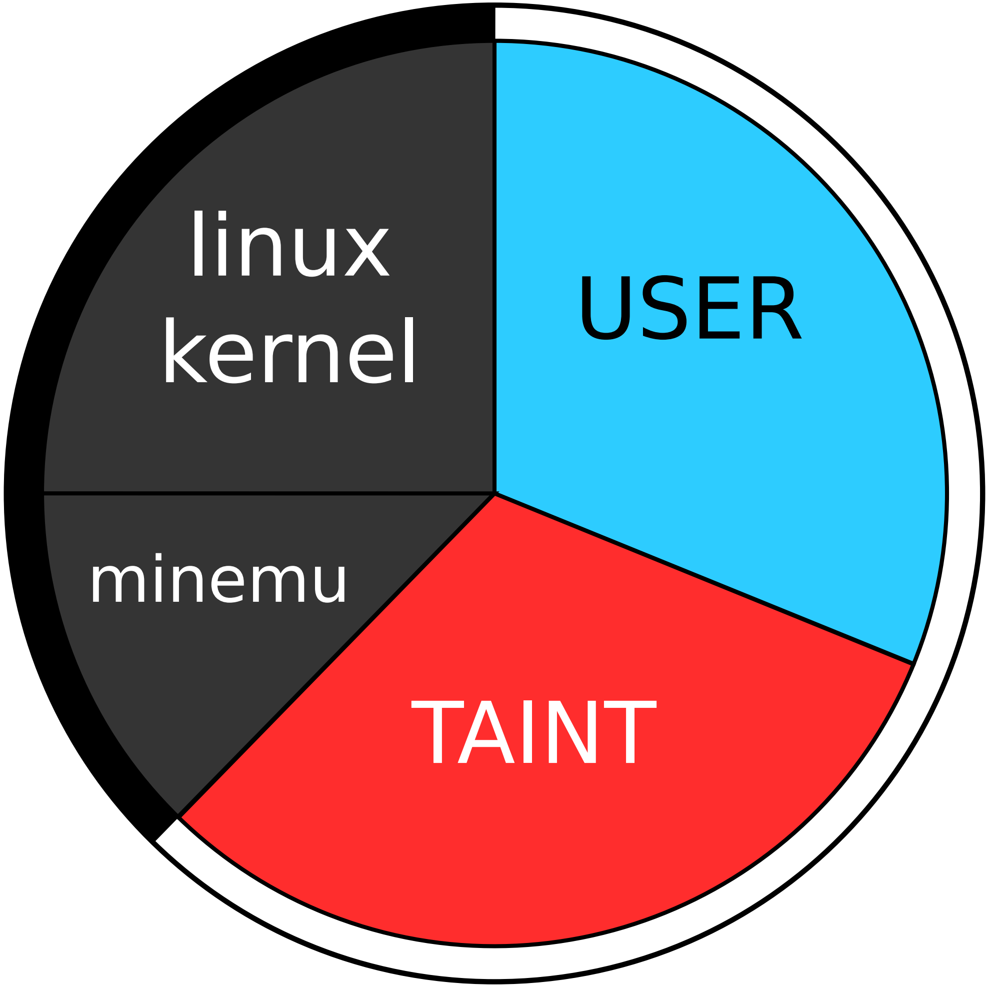
Memory layout (linux)



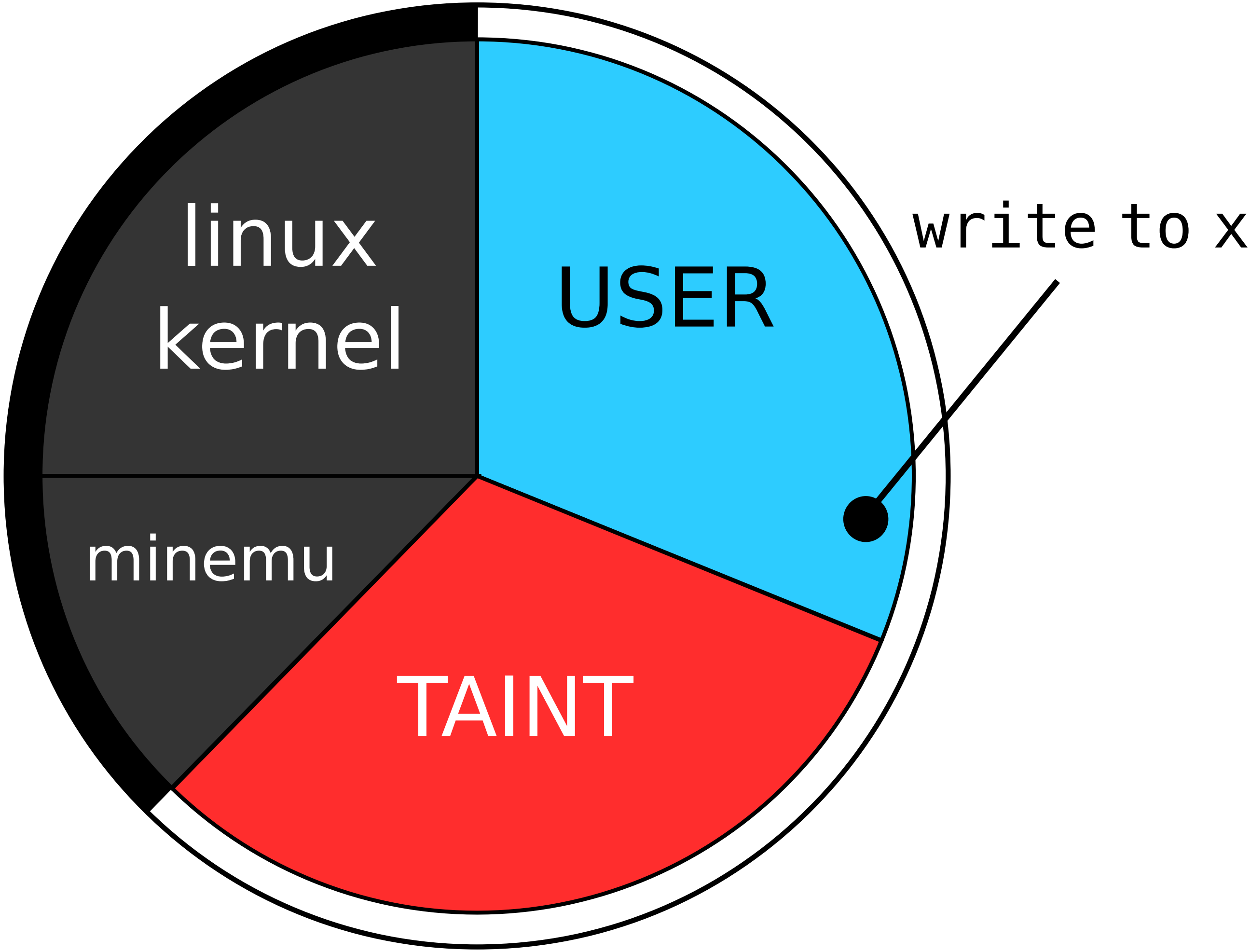
Memory layout (minemu)



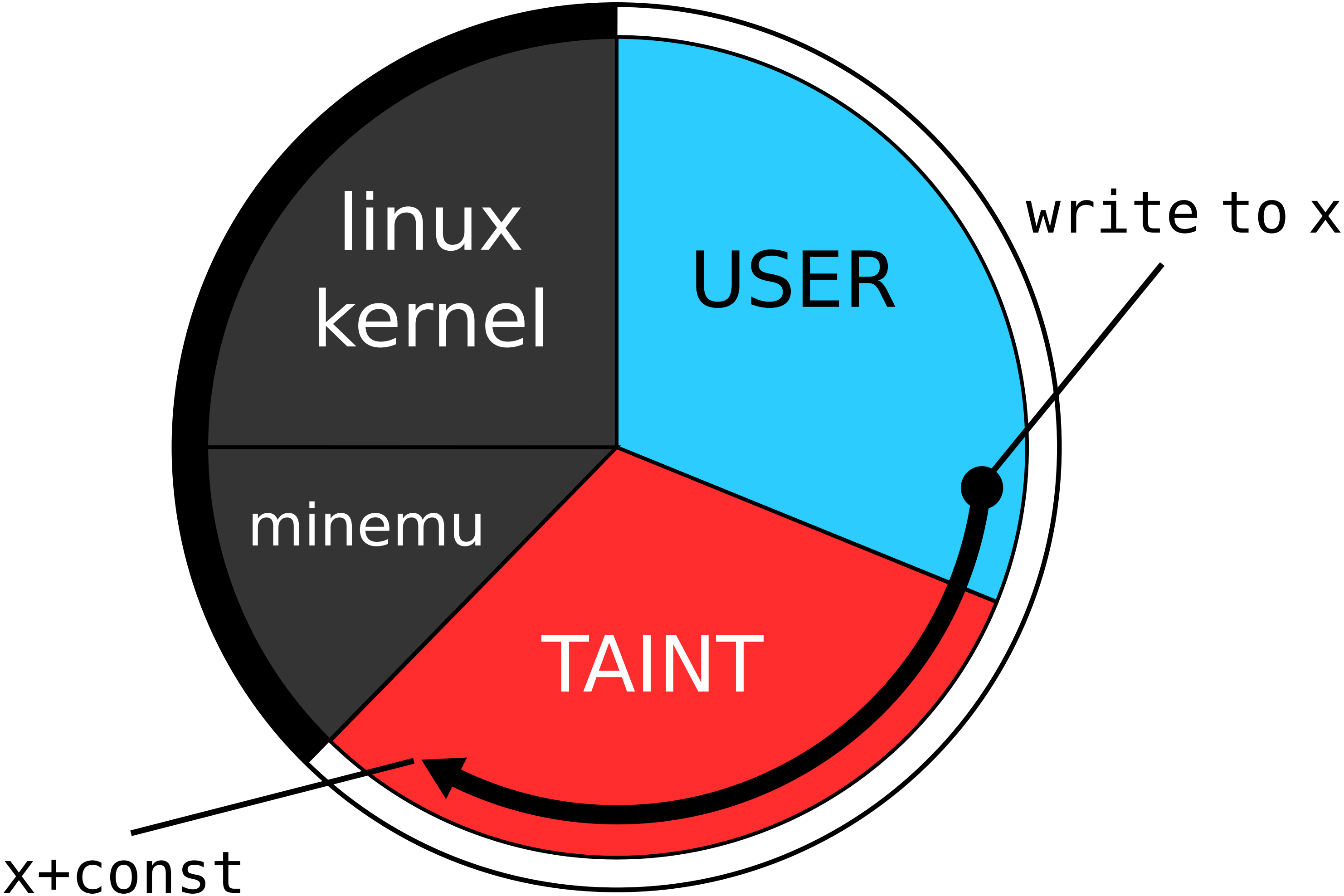
Memory layout (minemu)



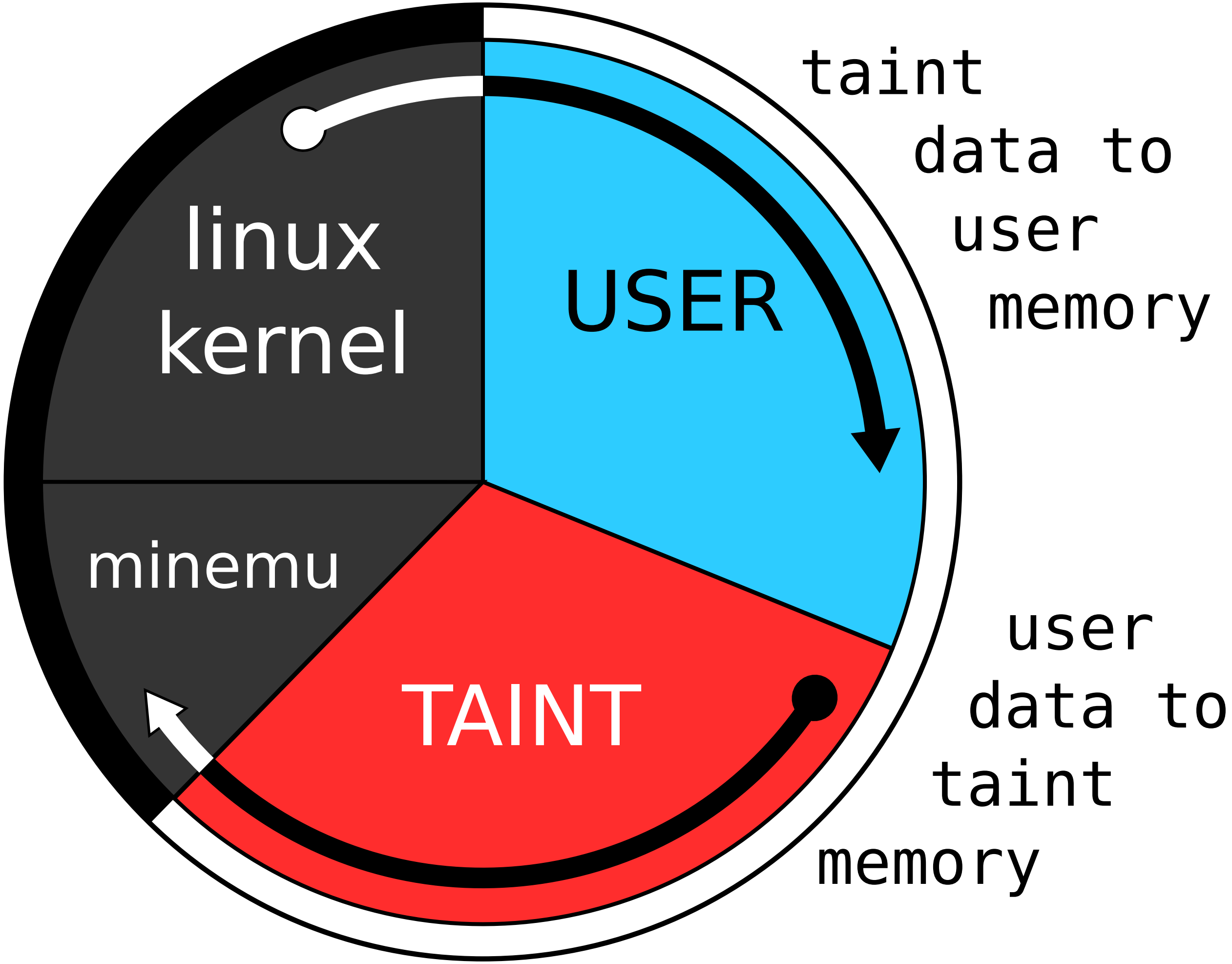
Memory layout (minemu)



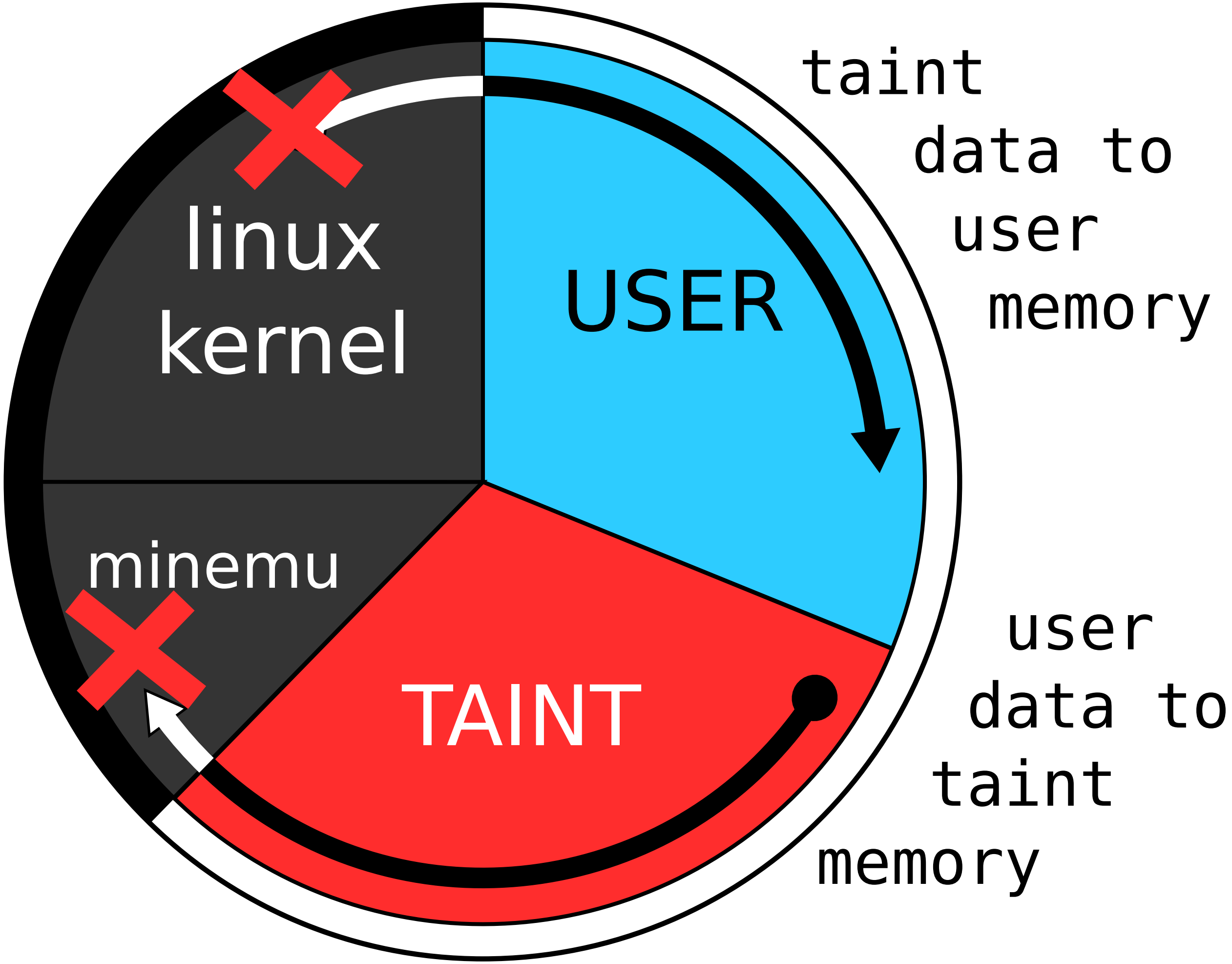
Memory layout (minemu)



Memory layout (minemu)



Memory layout (minemu)



Addressing shadow memory

```
mov EAX, (EDX)
```

Addressing shadow memory

```
mov EAX, (EDX)

address:

    EDX
```

Addressing shadow memory

```
mov EAX, (EDX)

address:

    EDX

taint:

    EDX+const
```

Addressing shadow memory

```
mov EAX, (EDX+EBX*4)
```

Addressing shadow memory

```
mov EAX, (EDX+EBX*4)

address:

    EDX+EBX*4
```

Addressing shadow memory

```
mov EAX, (EDX+EBX*4)

address:

    EDX+EBX*4

taint:

    EDX+EBX*4+const
```

Addressing shadow memory

```
push ESI
```

Addressing shadow memory

```
push ESI

address:

    ESP
```

Addressing shadow memory

```
push ESI

address:

    ESP

taint:

    ESP+const
```

Is this slowness fundamental?

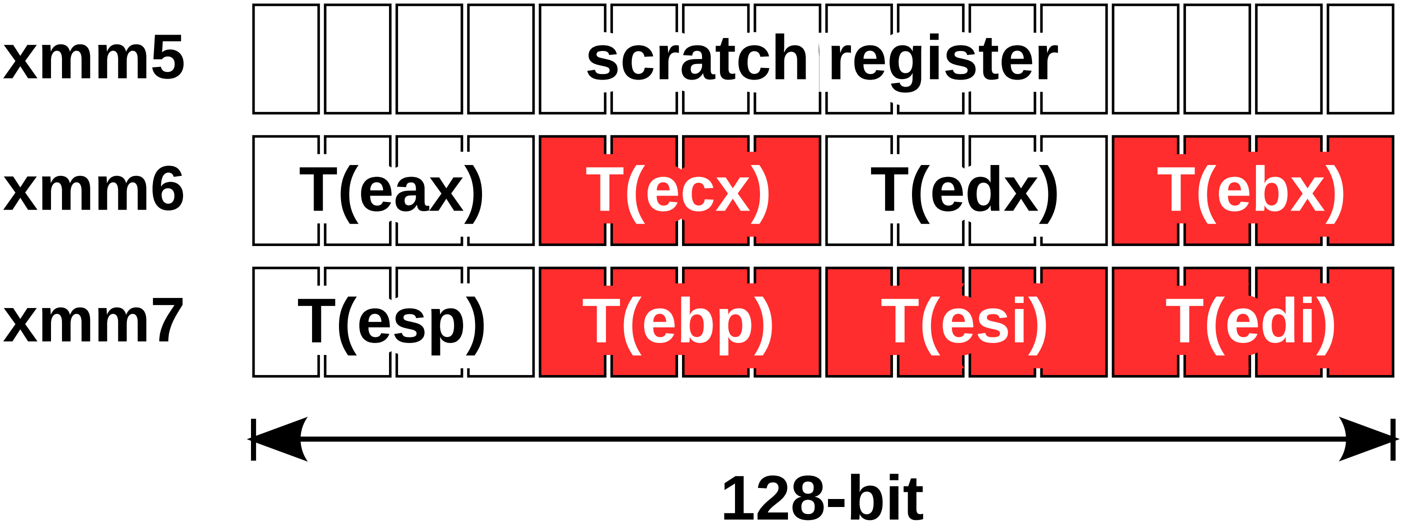
minemu

fast emulator

memory layout

▶ use SSE registers to hold taint

Taint propagation in SSE registers



xmm5 scratch register

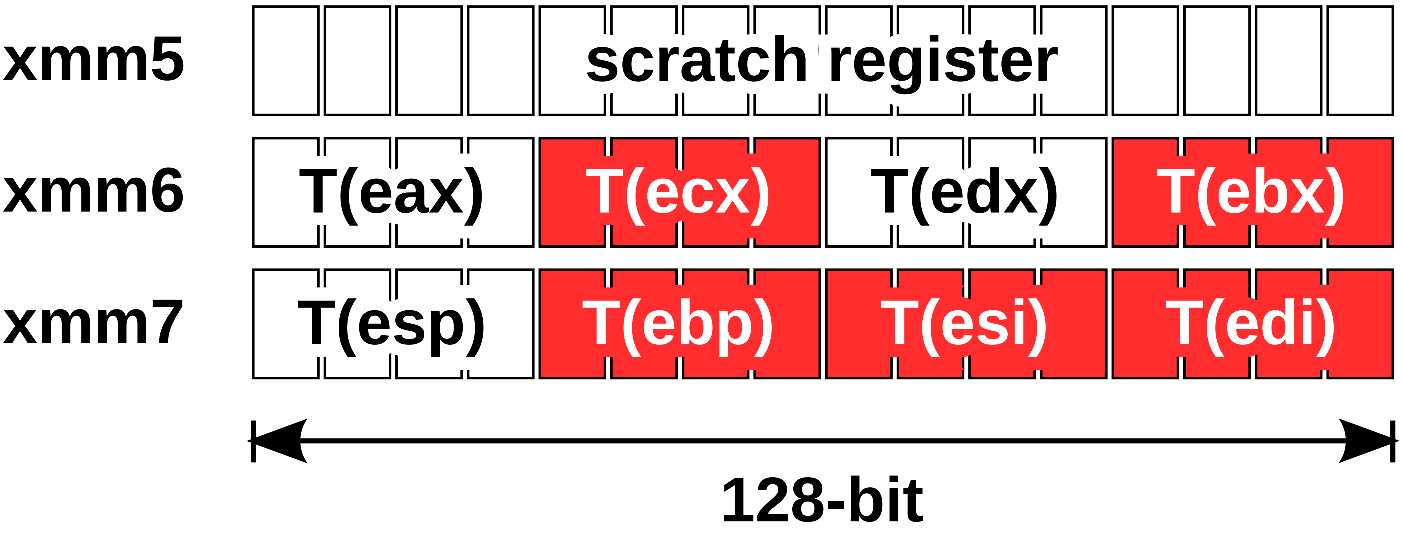
xmm6 T(eax) T(ecx) T(edx) T(ebx)

xmm7 T(esp) T(ebp) T(esi) T(edi)

128-bit

Taint propagation in SSE registers

```
add EDX, x
```



xmm5 scratch register

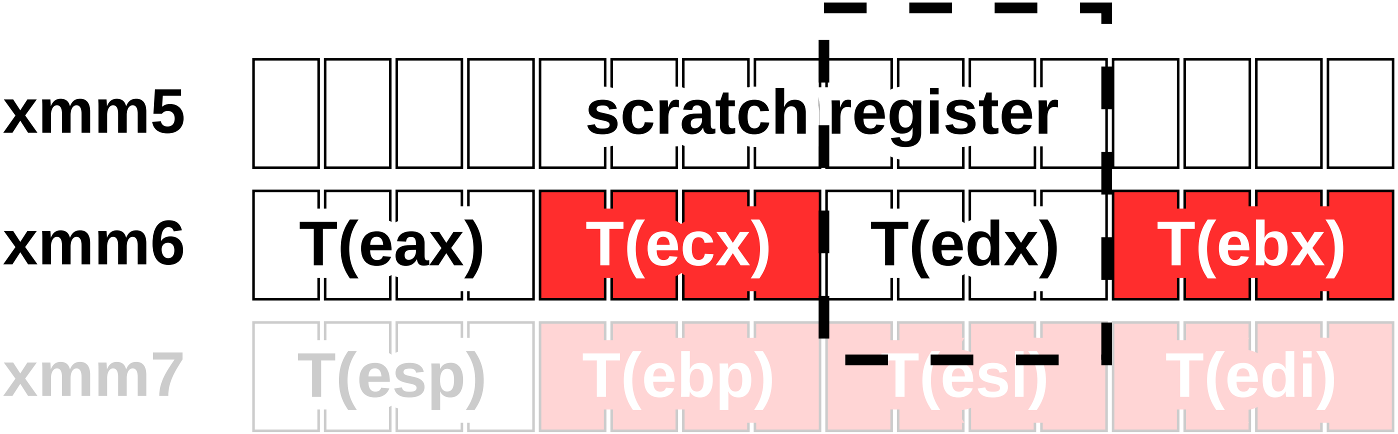
xmm6 T(eax) T(ecx) T(edx) T(ebx)

xmm7 T(esp) T(ebp) T(esi) T(edi)

128-bit

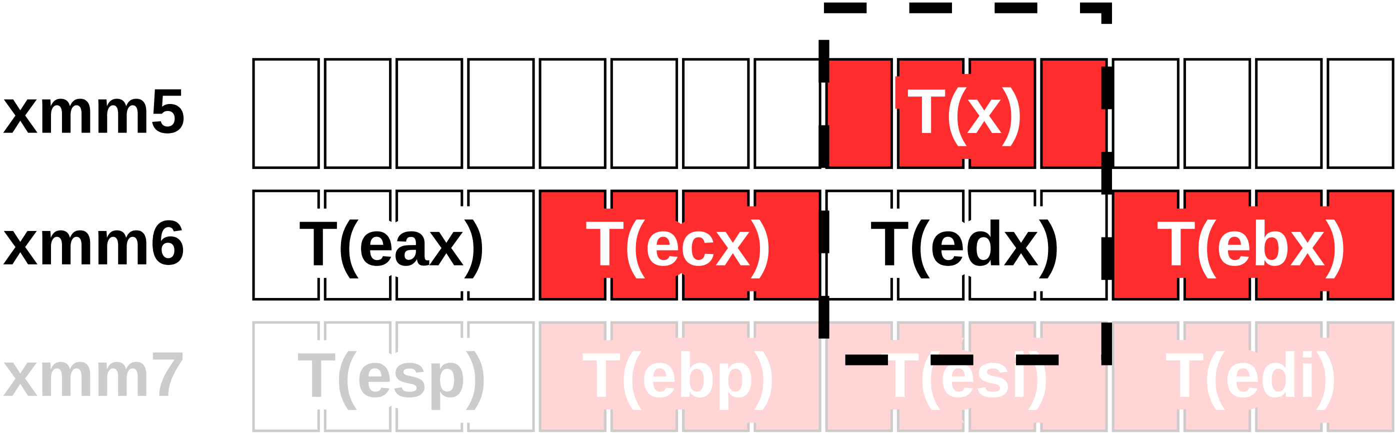
Taint propagation in SSE registers

```
add EDX, x
```



Taint propagation in SSE registers

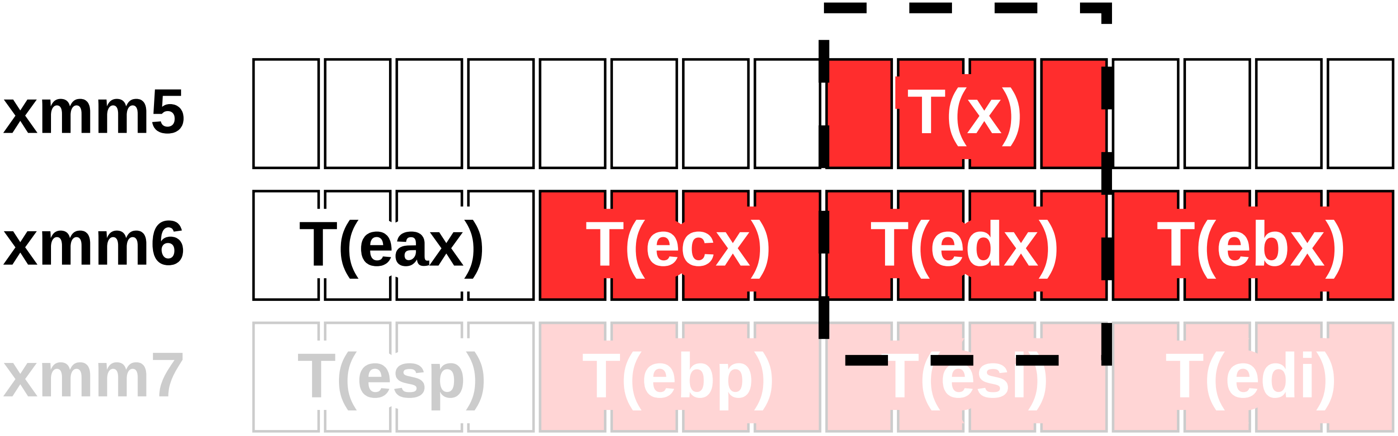
```
add EDX, x
```



```
vector insert
```

Taint propagation in SSE registers

`add EDX, x`

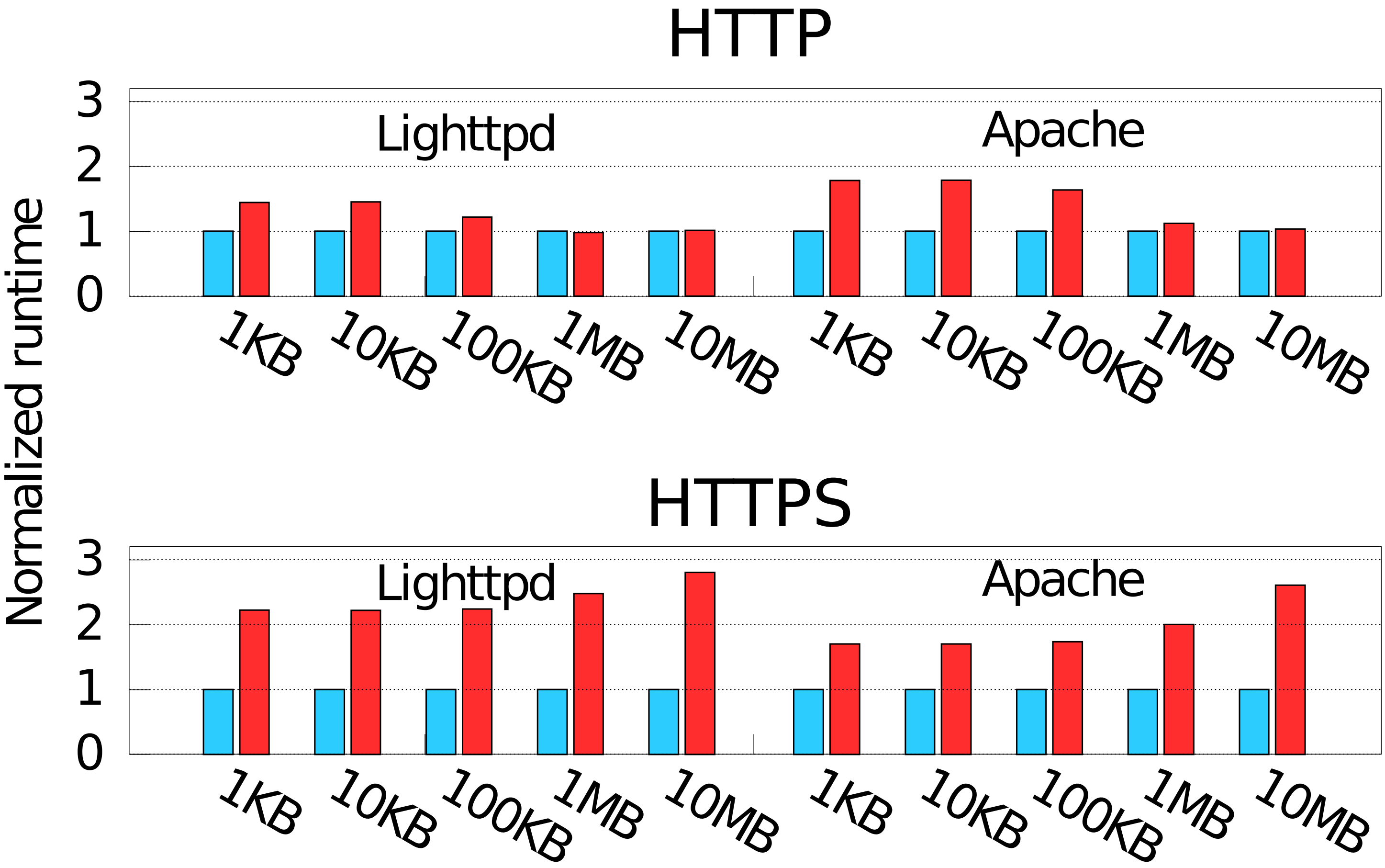


or

Effectiveness

Application	Type of vulnerability	Security advisory
Snort 2.4.0	Stack overflow	CVE-2005-3252
Cyrus imapd 2.3.2	Stack overflow	CVE-2006-2502
Samba 3.0.22	Heap overflow	CVE-2007-2446
Memcached 1.1.12	Heap overflow	CVE-2009-2415
Nginx 0.6.32	Buffer underrun	CVE-2009-2629
Proftpd 1.3.3a	Stack overflow	CVE-2010-4221
Samba 3.2.5	Heap overflow	CVE-2010-2063
Telnetd 1.6	Heap overflow	CVE-2011-4862
Ncompress 4.2.4	Stack overflow	CVE-2001-1413
Iwconfig V.26	Stack overflow	CVE-2003-0947
Aspell 0.50.5	Stack overflow	CVE-2004-0548
Htget 0.93	Stack overflow	CVE-2004-0852
Socat 1.4	Format string	CVE-2004-1484
Aeon 0.2a	Stack overflow	CVE-2005-1019
Exim 4.41	Stack overflow	EDB-ID#796
Htget 0.93	Stack overflow	
Tipxd 1.1.1	Format string	OSVDB-ID#12346

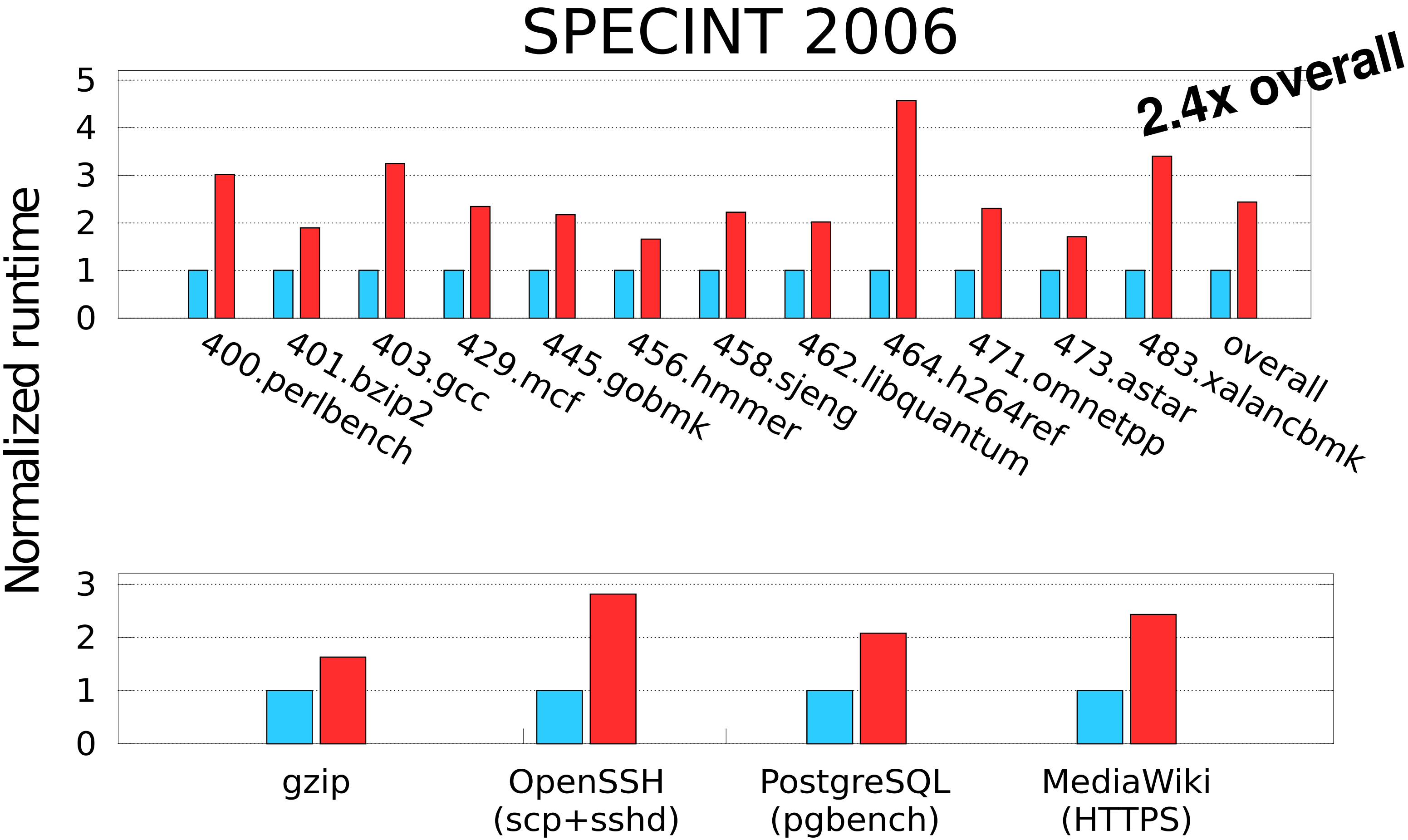
Performance



HTTP

HTTPS

Performance



Limitations

Limitations

Doesn't prevent memory corruption, only acts when the untrusted data is used for arbitrary code execution.

Limitations

Tainted pointer dereferences

```
tainted_pointer->some_field = useful_untainted_value;
```

Limitations

Tainted pointer dereferences

```
tainted_pointer->some_field = useful_untainted_value;
```

propagation can lead to false positives:

```
dispatch_table[checked_input]();
```

Limitations

Taint whitewashing

```
out = latin1_to_ascii[in];
```

Limitations

Format string attacks:

```
printf("%65534s %123$hn"); // Propagates taint in glibc

printf("FillerFiller...%123$hn"); // Does not :-(
```

Limitations

Does not protect against non-control-flow exploits

Limitations

Does not protect against non-control-flow exploits

```
void try_system(char *username, char *cmd)
{
    int user_rights = get_credentials(username);
    char buf[16] ; strcpy(buf, username);
    if (user_rights & ALLOW_SYSTEM)
        system(cmd);
    else
        log_error("user %s attempted login", buf);
}
```

Limitations

Does not protect against non-control-flow exploits

```
void try_system(char *username, char *cmd)
{
    int user_rights = get_credentials(username);
    char buf[16] ; strcpy(buf, username);
    if (user_rights & ALLOW_SYSTEM)
        system(cmd);
    else
        log_error("user %s attempted login", buf);
}
```

Limitations

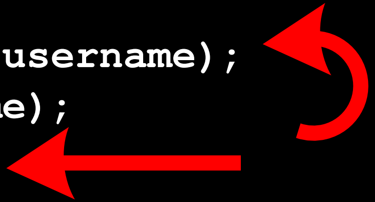
Does not protect against non-control-flow exploits

```
void try_system(char *username, char *cmd)
{
    int user_rights = get_credentials(username);
    char buf[16] ; strcpy(buf, username);
    if (user_rights & ALLOW_SYSTEM)
        system(cmd);
    else
        log_error("user %s attempted login", buf);
}
```

Limitations

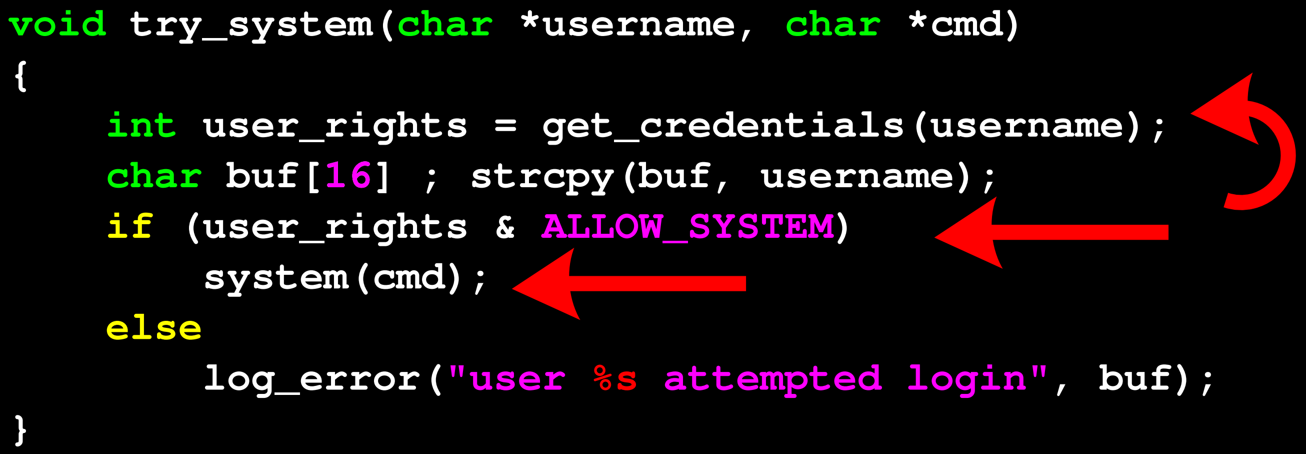
Does not protect against non-control-flow exploits

```c
void try_system(char *username, char *cmd)
{
    int user_rights = get_credentials(username);
    char buf[16] ; strcpy(buf, username);
    if (user_rights & ALLOW_SYSTEM)
        system(cmd);
    else
        log_error("user %s attempted login", buf);
}
```



Limitations

Does not protect against non-control-flow exploits



```
void try_system(char *username, char *cmd)
{
    int user_rights = get_credentials(username);
    char buf[16] ; strcpy(buf, username);
    if (user_rights & ALLOW_SYSTEM)
        system(cmd);
    else
        log_error("user %s attempted login", buf);
}
```

PROBLEM.php?-s

in some cases we can add validation hooks.

mysql_query() can be hooked to check for taint outside of literals in SQL queries.

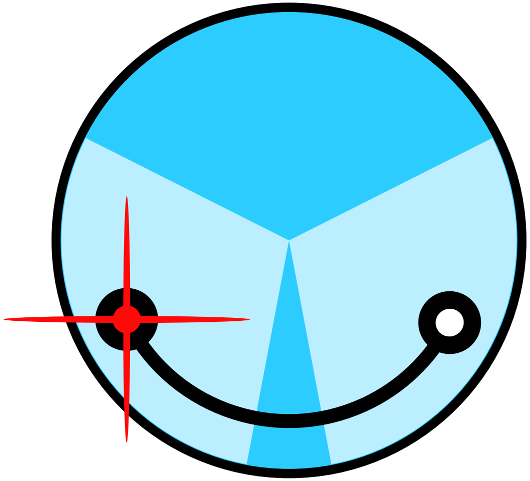
in some cases we can add validation hooks.

`mysql_query()` can be hooked to check for taint outside of literals in SQL queries.

`_IO_vfprintf()` in glibc can be hooked to check format strings for taint.

Demo

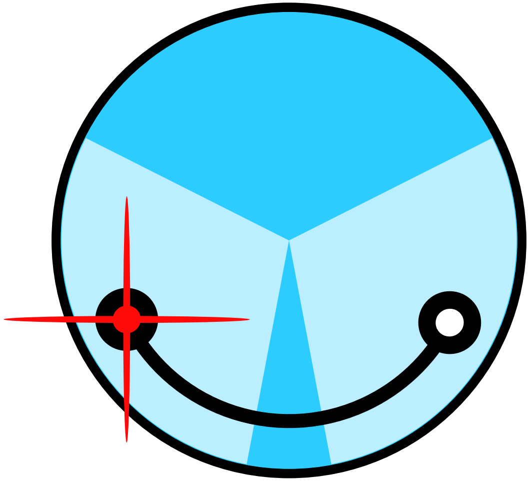
```
demo@demo:~# ./minemu bash
```



Minemu

```
git clone https://minemu.org/code/minemu.git
```

VU VRIJE UNIVERSITEIT AMSTERDAM

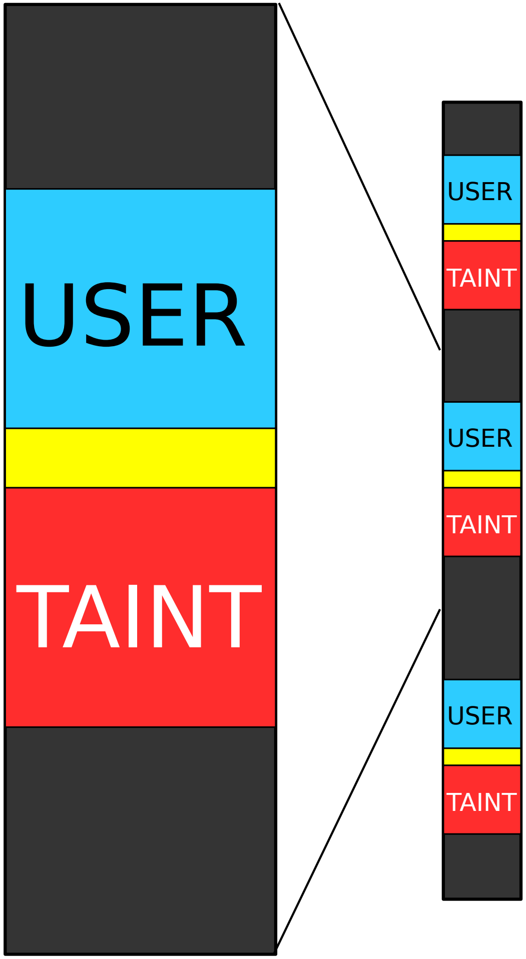


Minemu

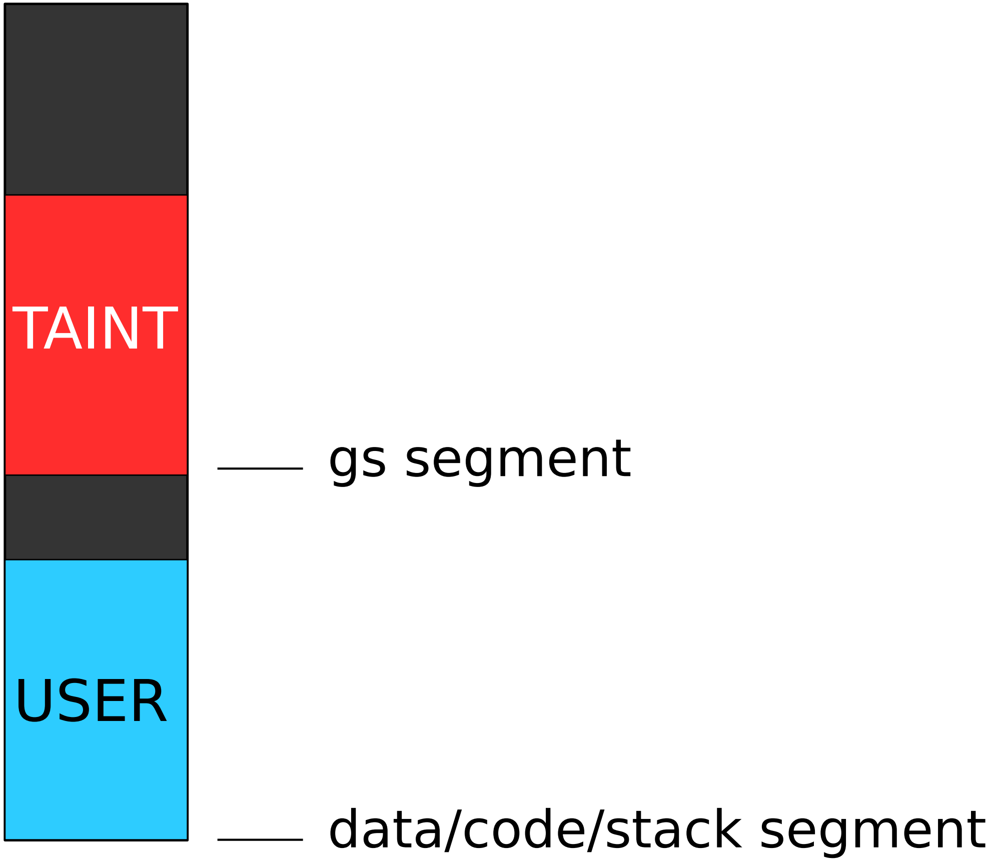
```
git clone https://minemu.org/code/minemu.git
```

any questions?

Memory layout (64 bit)



Memory layout (64 bit) alternative



TAINT
gs segment
USER
data/code/stack segment