Software Security: Buffer Overflow Attacks and Beyond

Tadayoshi Kohno

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Goals for Today

◆ Software security
  • Software lifecycle
  • Buffer overflow attacks
  • Other software security issues

◆ Lab 1 online on Wednesday
  • Make sure to attend section on Thursday!
  • Please form groups of 3 people

◆ Security Reviews / Current Events due Jan 25
  • Extra Credit if by Jan 18

◆ Impressed with the activity on the forums!!
Forum & Homework 1

- Goal: help develop the “security mindset”
- Best way to learn a foreign language: move to that country and immerse yourself in the language.
- Same thing applies to “security thinking”
- Forum: opportunity to think about security on a regular basis -- outside of class
  - Current events
  - New product announcements

- Also in forum:
  - Security in your everyday life
  - Movies / books
  - ...
Software Lifecycle (Simplified)

- Requirements
- Design
- Implementation
- Testing
- Use
Software problems are ubiquitous

Software Bug Halts F-22 Flight

Posted by kdawson on Sunday February 25, @06:35PM
from the dare-you-to-cross-this-line dept.

mgh02114 writes

"The new US stealth fighter, the [F-22 Raptor](#), was deployed for the first time to Asia earlier this month. On Feb. 11, twelve Raptors flying from Hawaii to Japan were forced to turn back when a software glitch crashed all of the F-22s' on-board computers as they crossed the international date line. The delay in arrival in Japan was [previously reported](#), with rumors of problems with the software. CNN television, however, this morning reported that every fighter completely lost all navigation and communications when they crossed the international date line. They reportedly had to turn around and follow their tankers by visual contact back to Hawaii. According to the CNN story, if they had not been with their tankers, or the weather had been bad, this would have been serious. CNN has not put up anything on their website yet."
Software problems are ubiquitous

1985-1987 -- **Therac-25 medical accelerator.** A radiation therapy device malfunctions and delivers lethal radiation doses at several medical facilities. Based upon a previous design, the Therac-25 was an "improved" therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. The Therac-25's X-rays were generated by smashing high-power electrons into a metal target positioned between the electron gun and the patient. A second "improvement" was the replacement of the older Therac-20's electromechanical safety interlocks with software control, a decision made because software was perceived to be more reliable.

What engineers didn't know was that both the 20 and the 25 were built upon an operating system that had been kludged together by a programmer with no formal training. Because of a subtle bug called a **race condition,** a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients die; others are seriously injured.

Software problems are ubiquitous

January 15, 1990 -- AT&T Network Outage. A bug in a new release of the software that controls AT&T's #4ESS long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines -- a message that the neighbors send out when they recover from a crash.

One day a switch in New York crashes and reboots, causing its neighboring switches to crash, then their neighbors' neighbors, and so on. Soon, 114 switches are crashing and rebooting every six seconds, leaving an estimated 60 thousand people without long distance service for nine hours. The fix: engineers load the previous software release.

Software problems are ubiquitous

- Other serious bugs (many others exist)
  - US Vicennes tracking software
  - MV-22 Osprey
  - Medtronic Model 8870 Software Application Card

From Exploiting Software and http://www.fda.gov/cdrh/recalls/recall-082404b-pressrelease.html
Adversarial Failures

- Software bugs are bad
  - Consequences can be serious

- Even worse when an intelligent adversary wishes to exploit them!
  - Intelligent adversaries: Force bugs into “worst possible” conditions/states
  - Intelligent adversaries: Pick their targets

- Buffer overflows bugs: Big class of bugs
  - Normal conditions: Can sometimes cause systems to fail
  - Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, ...)

Software bugs are bad

- Consequences can be serious

Even worse when an intelligent adversary wishes to exploit them!

- Intelligent adversaries: Force bugs into “worst possible” conditions/states
  - Intelligent adversaries: Pick their targets

Buffer overflows bugs: Big class of bugs

- Normal conditions: Can sometimes cause systems to fail
  - Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, ...)

Attacks

Buffer Overflows
A Bit of History: Morris Worm

- Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now an EECS professor at MIT
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- $10-100M worth of damage
Morris Worm and Buffer Overflow

One of the worm’s propagation techniques was a buffer overflow attack against a vulnerable version of fingerd on VAX systems

- By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
- Unable to determine remote OS version, worm also attacked fingerd on Suns running BSD, causing them to crash (instead of spawning a new copy)
More History

- Very common cause of Internet attacks
  - In 1998, over 50% of advisories published by CERT (computer security incident report team) were caused by buffer overflows

- Morris worm (1988): overflow in `fingerd`
  - 6,000 machines infected

  - 300,000 machines infected in 14 hours

- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!!)

- ...

- Still ubiquitous, especially in embedded systems
Attacks on Memory Buffers

- **Buffer** is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
    - If more data is stuffed into it, it spills into adjacent memory
  - If executable code is supplied as “data”, victim’s machine may be fooled into executing it – we’ll see how
    - Can give attacker control over machine
- First generation exploits: stack smashing
- Later generations: heaps, function pointers, off-by-one, format strings and heap management structures
Suppose Web server contains this function:

```c
void func(char *str) {
    char buf[126];
    ...
    strcpy(buf,str);
    ...
}
```

- **No bounds checking on `strcpy()`**
- **If `str` is longer than 126 bytes**
  - Program may crash
  - Attacker may change program behavior
Changing Flags

- Suppose Web server contains this function

  ```c
  void func(char *str) {
      int authenticated = 0;
      char buf[126];
      ...
      strcpy(buf, str);
      ...
  }
  ```

- **Authenticated** variable non-zero when user has extra privileges

- Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd
Memory Layout

- **Text region**: Executable code of the program
- **Heap**: Dynamically allocated data
- **Stack**: Local variables, function return addresses; grows and shrinks as functions are called and return

<table>
<thead>
<tr>
<th>Text region</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 0x00...0</td>
<td></td>
<td>Addr 0xFF...F</td>
</tr>
</tbody>
</table>
Stack Buffers

Suppose Web server contains this function

```c
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

When this function is invoked, a new **frame** with local variables is pushed onto the stack.

- Allocate local buffer (126 bytes reserved on stack)
- Copy argument into local buffer

Execute code at this address after func() finishes
What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack...

```c
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations

strcpy does NOT check whether the string at *str contains fewer than 126 characters.
Suppose buffer contains attacker-created string

- For example, *str contains a string received from the network as input to some network service daemon

When function exits, code in the buffer will be executed, giving attacker a shell

- Root shell if the victim program is setuid root
Buffer Overflow Issues

- Executable attack code is stored on stack, inside the buffer containing attacker’s string
  - Stack memory is supposed to contain only data, but...

- Overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will (probably) crash with segmentation violation
  - Attacker must correctly guess in which stack position his/her buffer will be when the function is called
Problem: No Range Checking

- strcpy does **not** check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “0” is encountered, ignoring the size of area allocated to buf

- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)
Does Range Checking Help?

- `strncpy(char *dest, const char *src, size_t n)`
  - If `strncpy` is used instead of `strcpy`, no more than `n` characters will be copied from `*src` to `*dest`
    - Programmer has to supply the right value of `n`

- Potential overflow in `htpasswd.c` (Apache 1.3):
  ```c
  strcpy(record, user);
  strcat(record, ":");
  strcat(record, cpw);
  ```

- Published “fix”:
  ```c
  ... strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record, ":");
  strcat(record, cpw, MAX_STRING_LEN-1);
  ```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")
Published “fix” for Apache htpasswd overflow:

```c
... strncpy(record, user, MAX_STRING_LEN-1);
strcat(record, ":");
strncat(record, cpw, MAX_STRING_LEN-1); ...
```

MAX_STRING_LEN bytes allocated for record buffer

- contents of *user
- : contents of *cpw

- Put up to MAX_STRING_LEN-1 characters into buffer
- Put “:”
- **Again** put up to MAX_STRING_LEN-1 characters into buffer
Off-By-One Overflow

Home-brewed range-checking string copy

```c
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

1-byte overflow: can’t change RET, but can change pointer to previous stack frame

- On little-endian architecture, make it point into buffer
- RET for previous function will be read from buffer!

This will copy 513 characters into buffer. Oops!
Stack Buffers

◆ Suppose Web server contains this function

```c
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

◆ When this function is invoked, a new frame with local variables is pushed onto the stack

Allocate local buffer (126 bytes reserved on stack)
Copy argument into local buffer

Execute code at this address after func() finishes
C uses **function pointers** for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(...)

---

**Function Pointer Overflow**

- Buffer with attacker-supplied input string
- Callback pointer
- Legitimate function F (elsewhere in memory)
Proper use of printf format string:

```c
int foo=1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

- This will print

    foo = 1234 in decimal, 4D2 in hex

Sloppy use of printf format string:

```c
char buf[14]="Hello, world!";
printf(buf);
// should’ve used printf("%s", buf);
```

- If buffer contains format symbols starting with %, location pointed to by printf’s internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf’s internal stack pointer.
Viewing Memory

- \%x format symbol tells printf to output data on stack

```c
... printf(“Here is an int:  \%x”,i); ...
```

- What if printf does \textbf{not} have an argument?

```c
... char buf[16]=“Here is an int:  \%x”;  
    printf(buf); ...
```

- Or what about:

```c
... char buf[16]=“Here is a string:  \%s”;  
    printf(buf); ...
```
Viewing Memory

- `%x` format symbol tells `printf` to output data on stack

```c
... printf("Here is an int:  %x",i); ...
```

- **What if `printf` does **not** have an argument?**

```c
... char buf[16]="Here is an int:  %x";
    printf(buf); ...
```

  - Stack location pointed to by `printf`’s internal stack pointer will be interpreted as an int.  (What if key, password, ...?)

- **Or what about:**

```c
... char buf[16]="Here is a string:  %s";
    printf(buf); ...
```

  - Stack location pointed to by `printf`’s internal stack pointer will be interpreted as a pointer to a string
%n format symbol tells printf to write the number of characters that have been printed

```c
... printf(“Overflow this!%n”, &myVar); ...
```

- Argument of printf is interpreted as destination address
- This writes 14 into myVar (“Overflow this!” has 14 characters)

What if printf does not have an argument?

```c
... char buf[16]=“Overflow this!%n”; printf(buf); ...
```

- Stack location pointed to by printf’s internal stack pointer will be interpreted as address into which the number of characters will be written.
Using %n to Mung Return Address

Buffer with attacker-supplied input string

“... attackString%n”, attack code

Number of characters in attackString must be equal to stack address where attack code starts

&RET

Somewhere on stack (on either side of RET)

C has a concise way of printing multiple symbols: %Mx will print exactly M bytes (taking them from the stack). If attackString contains enough “%Mx” so that its total length is equal to the most significant byte of the address of the attack code, this byte will be written into &RET. Repeat three times (four “%n” in total) to write into &RET+1, &RET+2, &RET+3, replacing RET with the address of attack code.

See “Exploiting Format String Vulnerabilities” for details
TOCTOU (Race Condition)

- TOCTOU == Time of Check to Time of Use

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

- Goal: Open only regular files (not symlink, etc)
- What can go wrong?
TOCTOU (Race Condition)

◆ TOCTOU == Time of Check to Time of Use

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

◆ Goal: Open only regular files (not symlink, etc)
◆ Attacker can change meaning of path between stat and open (and access files he or she shouldn’t)
Integer Overflow and Implicit Cast

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

```c
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```

**If** `len` **is negative, may copy huge amounts of input into** `buf`

(from www-inst.eecs.berkeley.edu—implflaws.pdf)
What if `len` is large (e.g., `len = 0xFFFFFFFF`)?

Then `len + 5 = 4` (on many platforms)

Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

```c
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```
Timing Attacks

- Assume there are no “typical” bugs in the software
  - No buffer overflow bugs
  - No format string vulnerabilities
  - Good choice of randomness
  - Good design
- The software may still be vulnerable to timing attacks
  - Software exhibits input-dependent timings
- Complex and hard to fully protect against
Password Checker

◆ Functional requirements
  • \texttt{PwdCheck(RealPwd, CandidatePwd)} should:
    – Return \texttt{TRUE} if \texttt{RealPwd} matches \texttt{CandidatePwd}
    – Return \texttt{FALSE} otherwise
  • \texttt{RealPwd} and \texttt{CandidatePwd} are both 8 characters long

◆ Implementation (like TENEX system)

\begin{verbatim}
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars
for i = 1 to 8 do
  if (RealPwd[i] != CandidatePwd[i]) then
    return FALSE
return TRUE
\end{verbatim}

◆ Clearly meets functional description
Attacker Model

\[
PwdCheck(\text{RealPwd}, \text{CandidatePwd}) \quad // \text{both 8 chars}
\]

\[
\text{for } i = 1 \text{ to } 8 \text{ do}
\]

\[
\text{if } (\text{RealPwd}[i] \neq \text{CandidatePwd}[i]) \text{ then}
\]

\[
\text{return FALSE}
\]

\[
\text{return TRUE}
\]

- Attacker can guess \text{CandidatePwds} through some standard interface
- Naive: Try all \(256^8 = 18,446,744,073,709,551,616\) possibilities
- Better: Time how long it takes to reject a CandidatePasswd. Then try all possibilities for first character, then second, then third, ....
- Total tries: \(256 \times 8 = 2048\)
Other Examples

Plenty of other examples of timings attacks

- AES cache misses
  - AES is the “Advanced Encryption Standard”
  - It is used in SSH, SSL, IPsec, PGP, ...
- RSA exponentiation time
  - RSA is a famous public-key encryption scheme
  - It’s also used in many cryptographic protocols and products
Randomness issues

- Many applications (especially security ones) require randomness
- Explicit uses:
  - Generate secret cryptographic keys
  - Generate random initialization vectors for encryption
- Other “non-obvious” uses:
  - Generate passwords for new users
  - Shuffle the order of votes (in an electronic voting machine)
  - Shuffle cards (for an online gambling site)
C’s rand() Function

C has a built-in random function: `rand()`

```c
unsigned long int next = 1;
/* rand: return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

Problem: don’t use `rand()` for security-critical applications!
- Given a few sample outputs, you can predict subsequent ones
PS3 Exploit

Today, January 3rd, George “Geohot” Hotz found and released the private root key for Sony’s Playstation 3 (PS3) video game console (http://www.geohot.com/). What this means is that homebrew software enthusiasts, scientists, and software pirates can now load arbitrary software on the PS3 and sign it using this key, and the system will execute it as trusted code. Legitimately, this allows Linux and other operating systems to take advantage of the PS3’s cell processor architecture; however, it also opens up avenues of software piracy previously impossible on Sony’s system without requiring any hardware modifications to the system (previous access of this kind required a USB hardware dongle).

How it Was Done

This was enabled by a cryptographic error by Sony developers in their update process. In the DSA signature algorithm, a number k is chosen from a supposedly random source for each signed message. So long as the numbers are unique, the system is secure, but duplicating a random number between messages can expose the private key to an untrusted party using simple mathematics (http://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/). Sony used the exact same “random value” k for all updates pushed to the system, making the signature scheme worthless.

The Most Secure

After Sony removed the “other OS” functionality of the PS3, greater scrutiny was placed on the PS3. Since it’s release in 2006, the Playstation 3 was considered the most secure of the three major video game consoles, as it was the only console without a “root” compromise in the four years since release (there were vulnerabilities limited to specific firmware or that required specialized hardware, but nothing that provided unfettered access). By comparison, Microsoft’s Xbox 360 was cracked over 4 years ago (http://www.theregister.co.uk/2007/03/01/xbox_hack), and the Wii was cracked over 2 years ago (http://wiibrew.org/wiki/Index.php).

Cullen Walsh
Mark Jordan
Peter Lipay
Problems in Practice

- One institution used (something like) `rand()` to generate passwords for new users
  - Given your password, you could predict the passwords of other users

  - Random number generator improperly seeded
  - Possible to trivially break into machines that rely upon Kerberos for authentication

- Online gambling websites
  - Random numbers to shuffle cards
  - Real money at stake
  - But what if poor choice of random numbers?
Big news... CNN, etc..
PS3 and Randomness

Example Current Event report from a past iteration of 484

- [https://catalyst.uw.edu/gopost/conversation/kohno/452868](https://catalyst.uw.edu/gopost/conversation/kohno/452868)
PS3 Exploit

Today, January 3rd, George “Geohot” Hotz found and released the private root key for Sony’s PlayStation 3 (PS3) video game console (http://www.geohot.com/). What this means is that homebrew software enthusiasts, scientists, and software pirates can now load arbitrary software on the PS3 and sign it using this key, and the system will execute it as trusted code. Legitimately, this allows Linux and other operating systems to take advantage of the PS3’s cell processor architecture; however, it also opens up avenues of software piracy previously impossible on Sony’s system without requiring any hardware modifications to the system (previous access of this kind required a USB hardware dongle).

How it Was Done

This was enabled by a cryptographic error by Sony developers in their update process. In the DSA signature algorithm, a number k is chosen from a supposedly random source for each signed message. So long as the numbers are unique, the system is secure, but duplicating a random number between messages can expose the private key to an untrusted party using simple mathematics (http://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/). Sony used the exact same “random value” k for all updates pushed to the system, making the signature scheme worthless.

The Most Secure

After Sony removed the “other OS” functionality of the PS3, greater scrutiny was placed on the PS3. Since it’s release in 2006, the Playstation 3 was considered the most secure of the three major video game consoles, as it was the only console without a “root” compromise in the four years since release (there were vulnerabilities limited to specific firmware or that required specialized hardware, but nothing that provided unfettered access). By comparison, Microsoft’s Xbox 360 was cracked over 4 years ago (http://www.theregister.co.uk/2007/03/01/xbox_hack), and the Wii was cracked over 2 years ago (http://wiibrew.org/wiki/Index.php).

Cullen Walsh
Mark Jordan
Peter Lipay
Other Problems

- **Key generation**
  - Ubuntu removed the randomness from SSL, creating vulnerable keys for thousands of users/servers
  - Undetected for 2 years (2006-2008)

- **Live CDs, diskless clients**
  - May boot up in same state every time

- **Virtual Machines**
  - Save state: Opportunity for attacker to inspect the pseudorandom number generator’s state
  - Restart: May use same “psuedorandom” value more than once
In the rush to clean up the Debian-OpenSSL fiasco, a number of other major security holes have been uncovered:

<table>
<thead>
<tr>
<th>Affected System</th>
<th>Security Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fedora Core</td>
<td>Vulnerable to certain decoder rings</td>
</tr>
<tr>
<td>Xandros (eee PC)</td>
<td>Gives root access if asked in stern voice</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Vulnerable to flattery</td>
</tr>
<tr>
<td>OLPC OS</td>
<td>Vulnerable to Jeff Goldblum’s Powerbook</td>
</tr>
<tr>
<td>Slackware</td>
<td>Gives root access if user says elvish word for “friend”</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>Turns out distro is actually just Windows Vista, with a few custom themes</td>
</tr>
</tbody>
</table>
DILBERT By Scott Adams

TOUR OF ACCOUNTING

OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.

NINE NINE NINE NINE NINE

ARE YOU SURE THAT'S RANDOM?

THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
Obtaining Pseudorandom Numbers

- For security applications, want “cryptographically secure pseudorandom numbers”
- Libraries include cryptographically secure pseudorandom number generators
- Linux:
  - /dev/random
  - /dev/urandom - nonblocking, possibly less entropy
- Internally:
  - Entropy pool gathered from multiple sources
Where do (good) random numbers come from?

- **Humans:** keyboard, mouse input
- **Timing:** interrupt firing, arrival of packets on the network interface
- **Physical processes:** unpredictable physical phenomena
Buffer overflow attacks

```assembly
void foo (char *argv[]) {
    push %ebp
    mov %esp,%ebp
    char buf[128];
    sub $0x88,%esp
    mov 0x8(%ebp),%eax
    strcpy(buf, argv[1]);
    add $0x4,%eax
    mov (%eax),%eax
    mov %eax,0x4(%esp)
    lea -0x80(%ebp),%eax
    mov %eax,(%esp)
    call 804838c <strcpy@plt>
}
leave
ret
```
How to defend against this?

```c
void foo (char *argv[]) {
    push    %ebp
    mov     %esp,%ebp
    char buf[128];
    sub     $0x88,%esp
    mov     0x8(%ebp),%eax
    strcpy(buf, argv[1]);
    add     $0x4,%eax
    mov     (%eax),%eax
    mov     %eax,0x4(%esp)
    lea     -0x80(%ebp),%eax
    mov     %eax,(%esp)
    call    804838c <strcpy@plt>
}
leave
ret
```
Stack Canary (StackGuard)

```c
void foo (char *argv[])
{
    int canary = <random>;
    char buf[128];
    strcpy(buf, argv[1]);
    assert(canary unchanged);
}
```

Any Canary Advice?

- Null byte stops `strcpy()` bugs
- CR-LF stops `gets()` bugs
- EOF stops `fread()` bugs
StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- PointGuard also places canaries next to function pointers and setjmp buffers
  - Worse performance penalty
- StackGuard doesn’t completely solve the problem (can be defeated)
Defeating StackGuard (Example, Sketch)

- Idea: overwrite pointer used by some `strcpy` and make it point to return address (RET) on stack
  - `strcpy` will write into RET without touching canary!

Suppose program contains `strcpy(dst,buf)`