Hands-on Ethical Hacking: Preventing & Writing Buffer Overflow Exploits

OWASP AppSec 2013

Rochester OWASP Chapter Lead

Ralph Durkee - Durkee Consulting, Inc.
info@rd1.net

Hands-on Ethical Hacking: Preventing & Writing Buffer Overflow Exploits

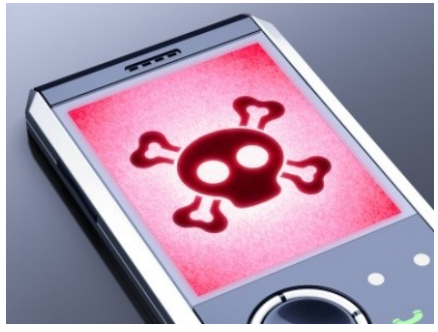
- **Please Get Started!**
- Boot-up your laptop
- Copy the VM folder to your laptop
- Start up Virtual box and boot up the VM
- Login – See Lab 0 handout

Ralph Durkee Background

- **Founder of Durkee Consulting** since 1996
- **Founder of Rochester OWASP** since 2004
- **President of Rochester ISSA Chapter**
- **Penetration Tester**, Security Trainer, Incident Handler and Auditor
- **Application Security**, development, auditing, PCI compliance, penetration testing and consulting
- **CIS (Center for Internet Security)** – development of benchmark security standards – Apache, Linux, BIND DNS, OpenLDAP, FreeRadius, Unix, FreeBSD

Agenda

- What is Ethical Hacking, and Why is it Important?
- Stack Overflows - What and How
- Lab 1 - Finding a Buffer Overflow
- The Intel Linux Stack Details
- Lab 2 - Finding the Return Pointer
- Lab 3 - Overwriting the Return Pointer
- Introduction to Shell Code
- Lab 4 – Shell Code 1
- Lab 5 – Full Exploit



Definition: Ethical Hacking

- **Hacking** – Manipulating things to do stuff beyond or contrary to what was intended by the designer or implementer.
- **Ethical Hacking** – Using hacking and attack techniques to find and exploit vulnerabilities for the purpose of improving security with the following:
  - Permission of the owners
  - In a professional and safe manner
  - Respecting privacy and property

Why Ethical Hacking?

- How Does Ethical Hacking Help?
- Why is it Important?
- Deeper Understanding of Vulnerabilities
- Deeper Understanding of Prevention Measures
- Understanding the Real Risk.
- Refuting Bogus Security Vendor Claims.
- Motivates Remediation!

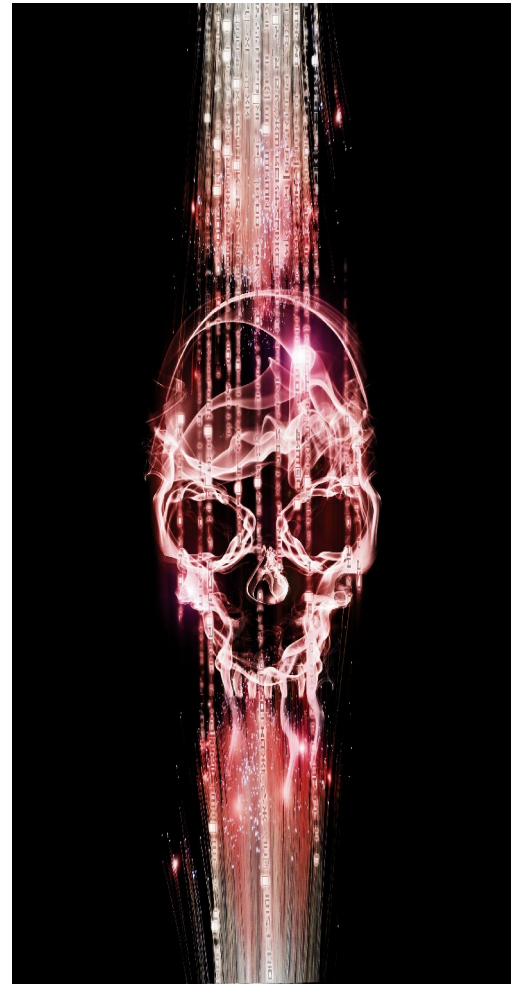
Buffer Overflow – Background

- Data Overruns Past End of the Buffer
- Buffers are allocated from
  - **Heap** – Dynamically allocated buffers
    free()/malloc() or New/Delete or other
  - **Stack** – local variables and function arguments
- Exploitation of Heap & Stack Overflows differs, as they have very different structures
- Our Focus – Stack Overflows

Stack Overflows – History

- 1988 Morris Worm – Unix Finger
- 2003 SQL Slammer Worm
  – MS SQL & MSDE
- 2003 Blaster Worm – DCOM RPC
- 2010 – Apple iOS Char String decoder
- Aug 2013 – Apache Santuario

Impact: Remote Code Execution



Finding Buffer Overflows

- Review the source code, if Available
- Search for dangerous functions
  - strcpy(3), strcat(3), sprintf(3), vsprintf(3) gets(3)
  - Better to use strncpy(3), strncat(3), snprintf(3), vsnprintf(3) fgets(3)
  - scanf(3), the *scand(3) if format length not limited
  - Getopt(3), strtok(3) and others.

Finding Buffer Overflows (2)

- What if you don't have source code?
- Search the binary executable with
  - strings(1)
  - debugger such as gdb(1) or OllyDbg(Windows)
- Metasploit
  - msfelsscan & msfpescan
  - Search for register usage and patterns

Finding Buffer Overflows (3)

- Fuzzing or Cramming Input
- Generates long repeating patterns attempting to overflow a buffer
- Wait for application crash or core dump
- Examine Registers,
  (especially the Instruction Pointer)
- Look for signs of the Fuzzed Data in the registers

Lab 1 – Finding Buffer Overflow

- See LAB 1 Handout

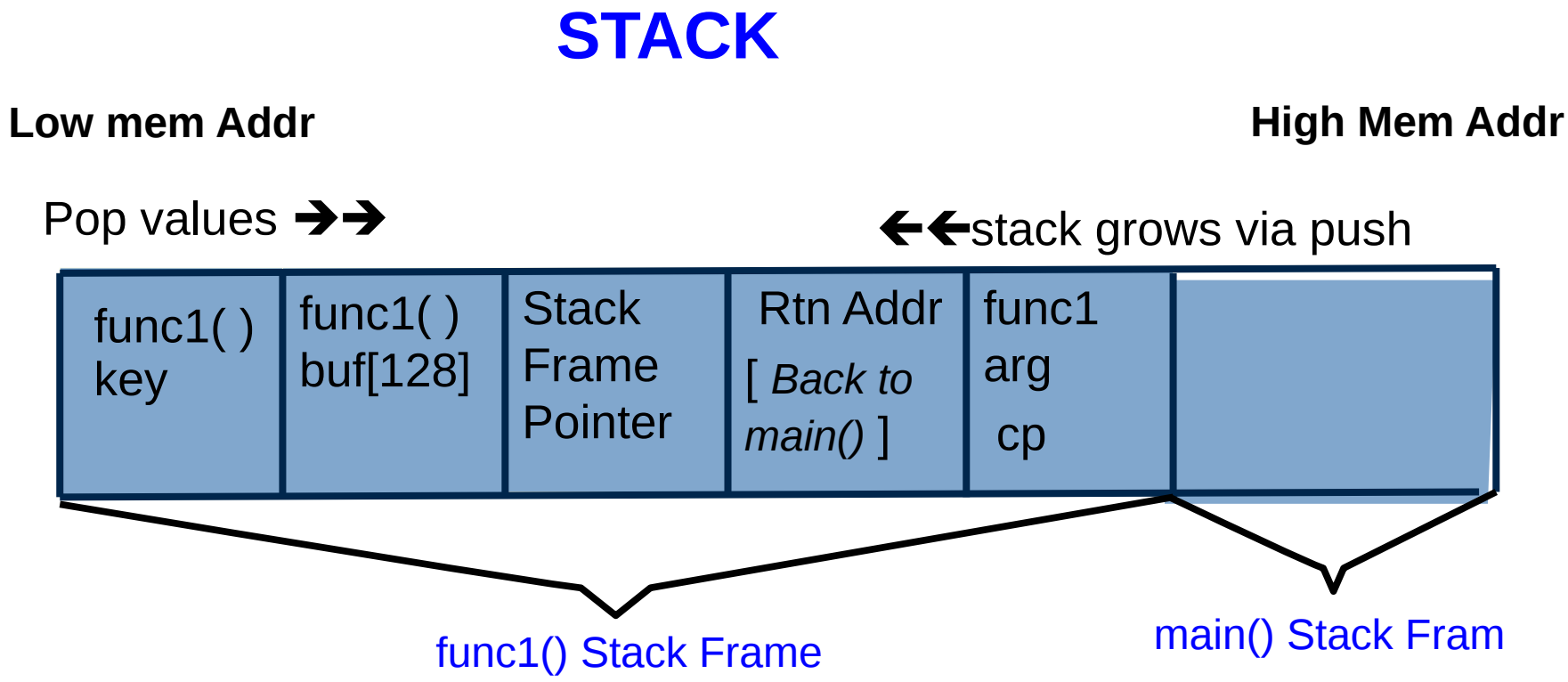
Stack Overflows – Example 1

- Sample Code – Stack Overflow

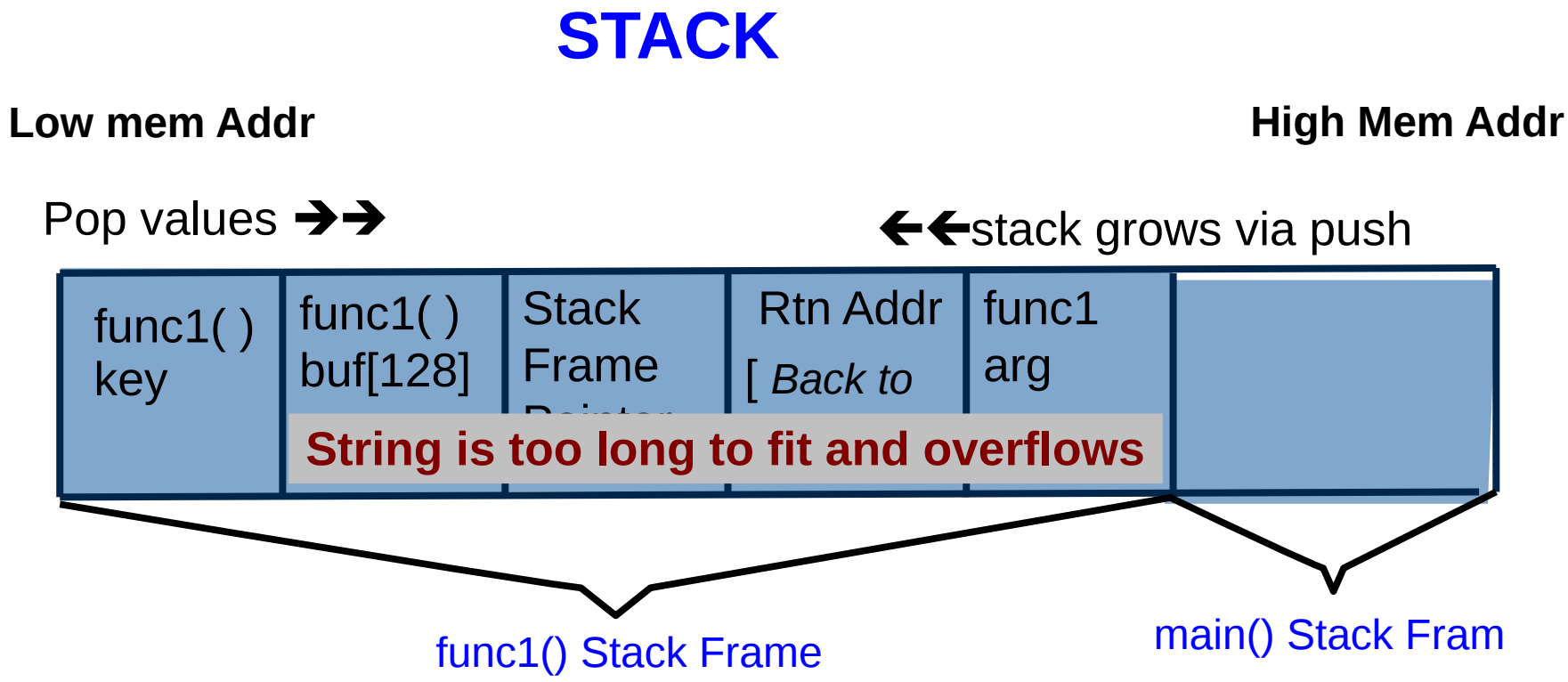
Terminal

```
#include <stdio.h>

void func1(char *cp)
{
    char buf[128];
    char * key = "ABCDEFGHIJKLMNOPQSTUVWXZ";
    strcpy(buf, cp);    // No limit in copy
}

main( int argc, char *argv[])
{
    func1(argv[1]);
    printf("\n");
    return(0);
}
                                                2,0-1         Top
```

The Stack with func1()



The Stack with func1() Overflow of buf[128]



So What's Accomplished?

func1() Stack frame

- Filled buf[128]
- Overwrote Stack Frame Pointer
- Overwrote the Return Address
- Modify Values of Arguments

main() Stack Frame

- **More of the same?**

- **Think like a Hacker:** How is this useful?

Intel x86 * Registers

16 bit	32 Bit	64 Bit	Description
AX	EAX	RAX	Accumulator
BX	EBX	RBX	Base Index (for arrays)
CX	ECX	RCX	Counter (for loops)
DX	EDX	RDX	Data / General
SP	ESP	RSP	Stack Pointer (top)
BP	EBP	RBP	Stack Base Pointer (current)
SI	ESI	RSI	Source Index (String ops)
DI	EDI	RDI	Destination Index (String ops)
IP	EIP	RIP	Instruction Pointer

Stack/Heap Platform Differences

- **Meta Data Structure Details for Stack and Heap vary significantly**
- **Different platforms**: Windows Different from Linux
- **Releases**: Different for Windows versions or Linux Distros
- **Usage**: C++ new/delete has different heap from the C malloc/free
- **Compiler**: Even different based on compiler used and compiler options used.

How to Write a Stack Overflow Exploit

1. Find input that causes a stack overflow
2. Find the data location that overwrites the return address
3. Write or borrow the Shell code payload for the exploit, and include it in the input
4. Find the approximate address where the payload lands
5. Overwrite return pointer with the correct address
6. Document, Report and Bask in the Glory! :-)

2. Finding the Return Pointer

- **What portion of the malicious input overwrite the return pointer?**
- **Techniques:**
  - Input a repeating Pattern like: AAAAAAAAAABBBBBBBBBCCCCCCCCDDDDDDDDDDEEEEEEEEE …
  - Exploit in gdb and Check the EIP/RIP value.
  - Then use a differentiating pattern like: ABCEDFGHIJABCEDFGHIJABCEDFGHIJABCEDFGHIJABCEDFGHIJ ….
  - Intersection of the pattern determines the location.

Lab 2 - Finding the Return Pointer

- See Lab 2 Handout

Lab 3 - Overwriting the Return Pointer

- See Lab 3 Handout

Shell Code Introduction

- Shell Code = payload (what do you want to do?)
- Most common is to execute a shell
- Other options:
  - Add Admin user account
  - Connect to a network port, and read commands
  - Listen on network for commands
  - Reset an admin password
  - What ever is useful for an attack
-

Shell Code Short Cut

Quick Approach, write a shell function

```
shell()
{
    execve("/bin/bash",NULL,NULL);
}
```

Disassemble the function

(gdb) disassemble shell
Dump of assembler code for function shell:

0x0804842c <+0>: push %ebp

0x0804842d <+1>: mov %esp,%ebp

0x0804842f <+3>: sub $0x18,%esp

0x08048432 <+6>: movl $0x0,0x8(%esp)

0x0804843a <+14>:movl $0x0,0x4(%esp)

0x08048442 <+22>: movl $**0x8048526**,(%esp)

0x08048449 <+29>: call 0x8048350 <execve@plt>

Lab 4 - Shell Code Short Cut

- See Lab 4 Handout

Shell Code Requirements

Shell code requirements

- Written in Assembler
- Specific Operating System (Mac. Windows, Linux, Unix)
- Specific to the Hardware (x86, x86_64, Sparc, RISC etc)
- Must fit into the vulnerable buffer space
- Often can't contain zero bytes
- Must have position independent code

Exploit & Shell Code Resources

Where to get Shell Code

- www.exploit-db.com/shellcode/
- Metasploit
- www.securityfocus.com
- Google
- http://www.rapid7.com/db/
- Others

Lab 5 – Full Exploit

- Lab 5 Handout

Summary: Hands-On Ethical Hacking - Stack Overflows

- Ethical Hacking – **Always with Permission & Stay in Scope**
- Understanding how it works leads to understanding how it breaks
- Experimentation and Failure are essential
- Hacking = Learning
- Test all assumptions
- Don't boast or get a big ego, but do have Fun!

Thank You!

Ralph Durkee
info@rd1.net

Resources - Non-Profit Groups & Events

Rochester ISSA Chapter

http://RocISSA.org

OWASP Rochester Chapter Information

https://www.OWASP.org/rochester

Rochester Security Summit

htttps://RochesterSecurity.org