Application security

September 25, 2015

Administrative – submittal instructions

- answer the lab assignment's questions in written report form, as a text, pdf, or Word document file (no obscure formats please)
- email to csci530l@usc.edu
- exact subject title must be "applicationsecuritylab"
- deadline is start of your lab session the following week
- reports not accepted (zero for lab) if
  - late
  - you did not attend the lab (except DEN or prior arrangement)
  - email subject title deviates

Administrative – DETER accounts

- accounts were created last night
- you should have received an auto-generated email message with information and instructions
- please read it and respond to its request to act (change password) within 3 days

Upcoming lab schedule adjustment

- we will have a capture-the-flag exercise
- among competitive teams
- extending over 5 weeks
- after the midterm, 10/16
- with the help of Prof Jelena Mirkovic
- it will replace one of our scheduled labs
- more to be announced later

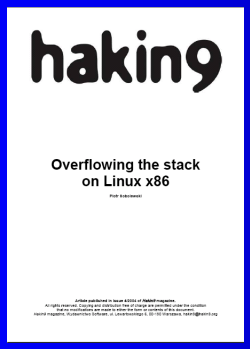
Administrative

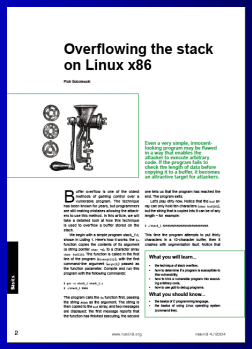
- refer during upcoming lab to these slides' screenshots
  - recommend you have paper or electronic access to those slides that contain detailed screenshots (lab asks you to mimic screenshot activities)
- use *only* the provided VM environment

  (CentOS 4.3 min-gdb)
  - it has been customized a little
  - other platforms/compilers generally *won't work*

Administrative – pre-homework

- advance preparation for this lab
- read through page 8

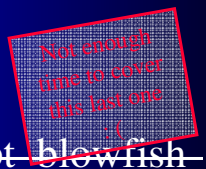




http://sobolewscy.in5.pl/piotr/publikacje/hakin9/stackoverflow_en.pdf
http://www-scf.usc.edu/~csci530l/downloads/stackoverflow_en.pdf

Three aspects of lab

- Generic stack overflow
- heartbleed bounds checking oversight



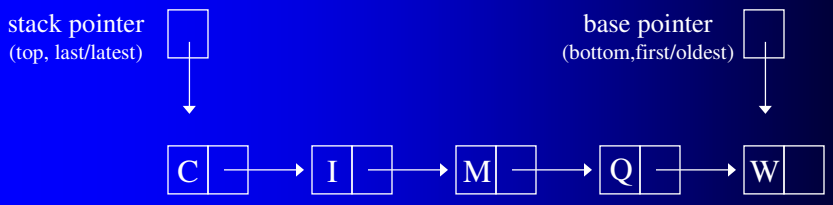
- ~~sign extension code flaw in crypt_blowfish~~

Generic stack buffer overflow

Stack buffer overflow

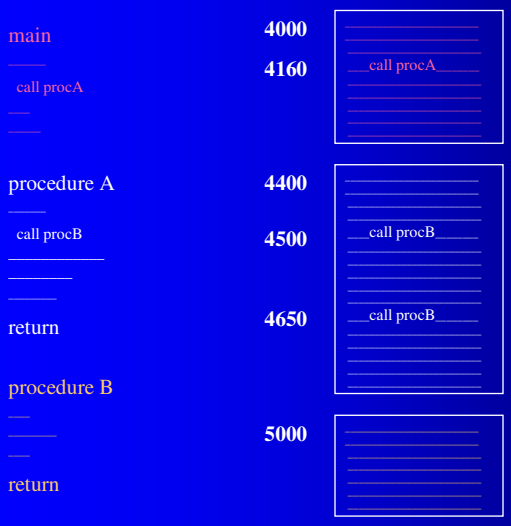
- what's a stack?
- what's an overflow?

Special list: a stack

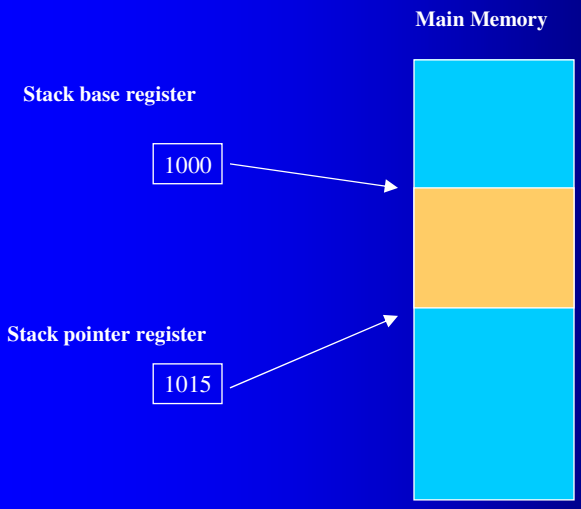


All insertions and deletions occur at one end, the "top.

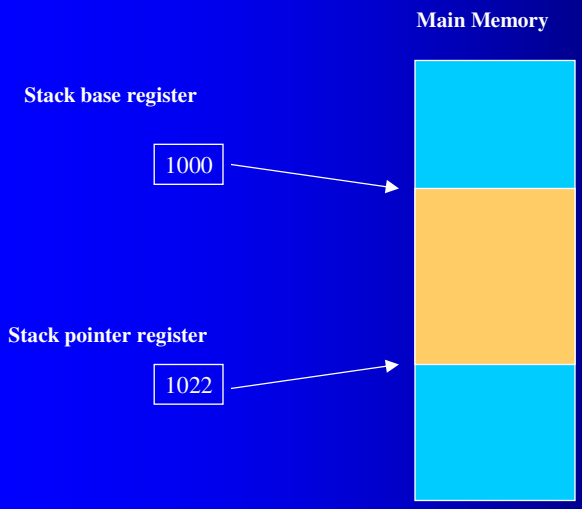
Used for intra-program control flow



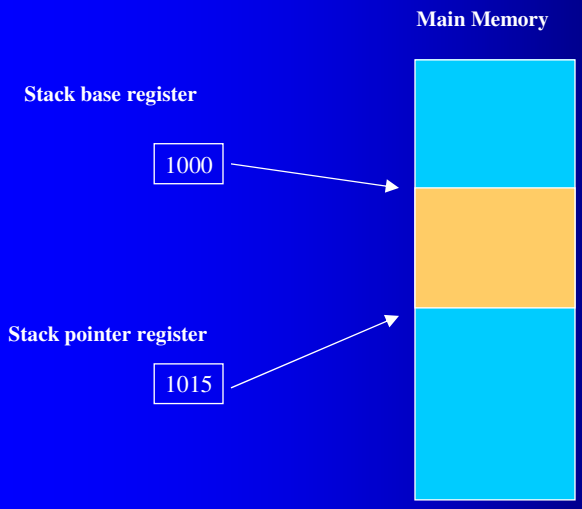
Uses a stack to get back



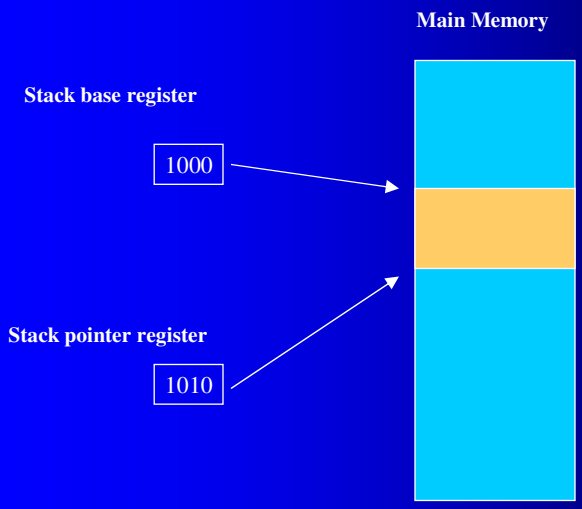
Uses a stack



Uses a stack



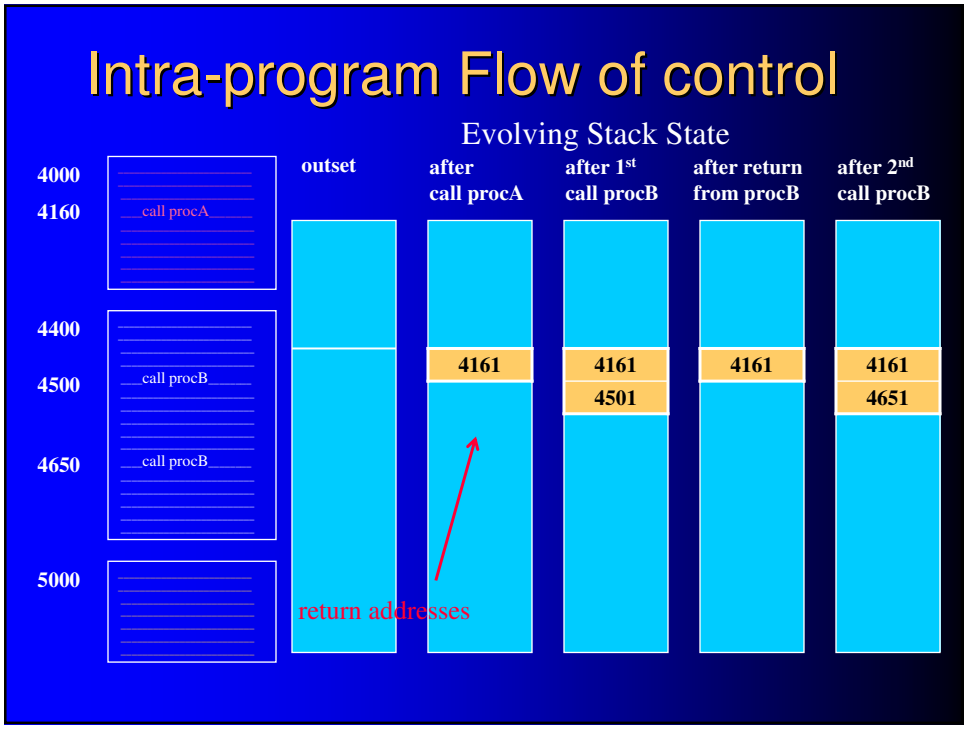
Uses a stack



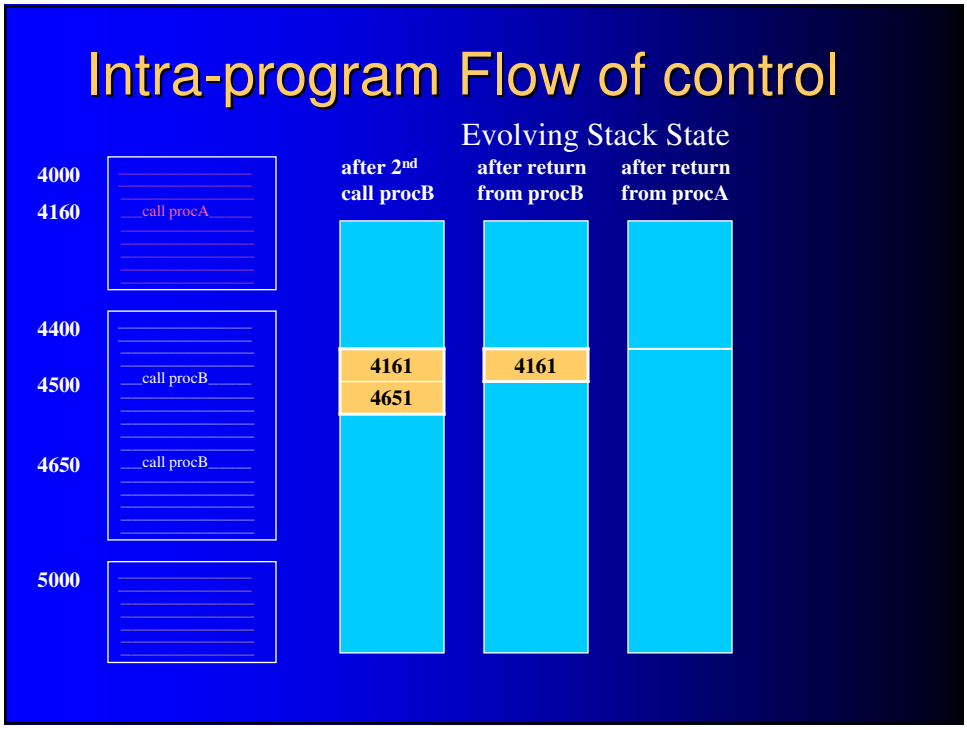
Finding your way back—

breadcrumbs & return addresses

Intra-program Flow of control



Intra-program Flow of control

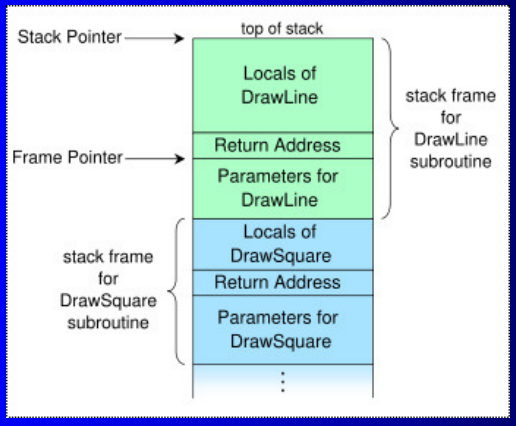


Not only return addresses get “stacked”

- local variables
- frame (intrastack) pointers
- return addresses
- arguments/parameters for called functions

Stack diagram

“For example, if a subroutine named DrawLine is currently running, having just been called by a subroutine DrawSquare, the top part of the call stack might be laid out like this (where the stack is growing towards the top):

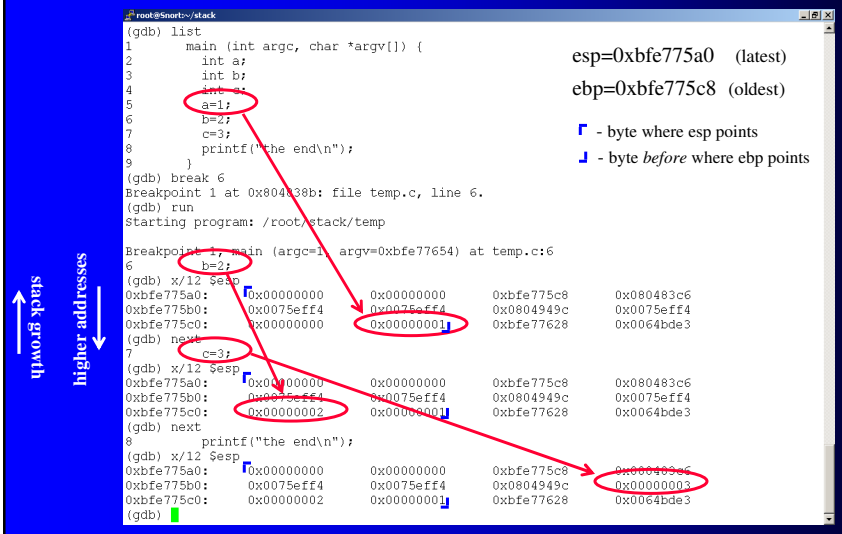


From: http://en.wikipedia.org/wiki/Stack_frame#Structure

vars.c – has local variables

```
main (int argc, char *argv[]) {
   int a;
   int b;
   int c;
   a=1;
   b=2;
   c=3;
   printf("the end\n");
}
```

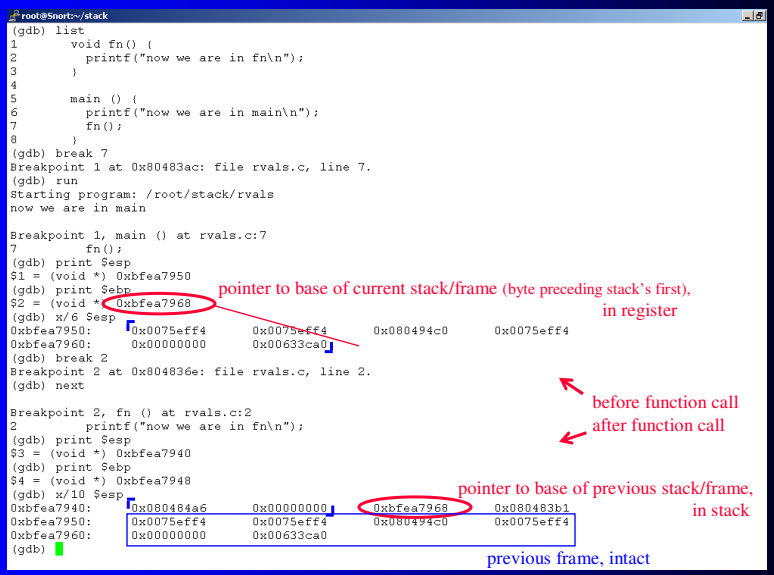
Local variables on the stack



rvals.c – has a function

```
void fn() {
  printf("now we are in fn\n");
}

main () {
  printf("now we are in main\n");
  fn();
}
```

frame pointers & return addresses



…continued… & return addresses

```
(gdb) print $esp
$3 = (void *) 0xbfea7940
(gdb) print $ebp
$4 = (void *) 0xbfea7948
(gdb) x/10 $esp
0xbfea7940:     0x080484a6      0x00000000      0xbfea7968      0x080483b1
0xbfea7950:     0x0075eff4      0x0075eff4      0x080494c0      0x0075eff4
0xbfea7960:     0x00000000      0x00633ca0
(gdb) disas main
Dump of assembler code for function main:
0x08048380 <main+0>:    push   %ebp
0x08048381 <main+1>:    mov    %esp,%ebp
0x08048383 <main+3>:    sub    $0x8,%esp
0x08048386 <main+6>:    and    $0xfffffff0,%esp
0x08048389 <main+9>:    mov    $0x0,%eax
0x0804838e <main+14>:   add    $0xf,%eax
0x08048391 <main+17>:   add    $0xf,%eax
0x08048394 <main+20>:   shr    $0x4,%eax
0x08048397 <main+23>:   shl    $0x4,%eax
0x0804839a <main+26>:   sub    %eax,%esp
0x0804839c <main+28>:   sub    $0xc,%esp
0x0804839f <main+31>:   push   $0x80484a6
0x080483a4 <main+36>:   call   0x80482b0
0x080483a9 <main+41>:   add    $0x10,%esp
0x080483ac <main+44>:   call   0x8048368 <fn>
0x080483b1 <main+49>:   leave
0x080483b2 <main+50>:   ret
End of assembler dump.
(gdb) print $ebp+4
$5 = (void *) 0xbfea794c
(gdb) x 0xbfea794c
0xbfea794c:     0x080483b1
(gdb)
```

breadcrumb!

place to go back
to in calling routine, when done

where to go back to

where you left off

stack_2.c –function parameters to pass

```
void fn(int arg1, int arg2) {
  int x; int y;
  x=3; y=4;
  printf("now we are in fn\n");
}

main () {
  int a; int b;
  a=1; b=2;
  fn(a, b);
}
```

…and args for called functions

```
1    void fn(int arg1, int arg2) {
2      int x; int y;
3      x=3; y=4;
4      printf("now we are in fn\n");
5    }
6
7    main () {
8      int a; int b;
9      a=1; b=2;
10     fn(a, b);
11   }
(gdb) break 10
Breakpoint 1 at 0x80483b8: file stack_2.c, line 10.
(gdb) break 4
Breakpoint 2 at 0x804837c: file stack_2.c, line 4.
(gdb) run
Starting program: /root/stack/stack_2

Breakpoint 1, main () at stack_2.c:10
10     fn(a, b);
(gdb) print $esp
$1 = (void *) 0xbfed9d10
(gdb) print $ebp
$2 = (void *) 0xbfed9d28
(gdb) x/6 $esp
0xbfed9d10:     0x0075eff4      0x0075eff4      0x080494c4      0x0075eff4
0xbfed9d20:     0x00000002      0x00000001
(gdb) next

Breakpoint 2, fn (arg1=1, arg2=2) at stack_2.c:4
4      printf("now we are in fn\n");
(gdb) print $esp
$3 = (void *) 0xbfed9cf0
(gdb) print $ebp
$4 = (void *) 0xbfed9cf8
(gdb) x/14 $esp
0xbfed9cf0:     0x00000004      0x00000003      0xbfed9d28      0x080483c6
0xbfed9d00:     0x00000001      0x00000002      0xbfed9d28      0x080483e6
0xbfed9d10:     0x0075eff4      0x0075eff4      0x080494c4      0x0075eff4
0xbfed9d20:     0x00000002      0x00000001
```

args for fn, placed on stack via main

local vars of main (bottom) and fn (top)

pointer to base of previous stack frame

return address

…continued

```
(gdb) print $esp
$3 = (void *) 0xbfed9cf0
(gdb) print $ebp
$4 = (void *) 0xbfed9cf8
(gdb) x/14 $esp
0xbfed9cf0:     0x00000004      0x00000003      0xbfed9d28      0x080483c6
0xbfed9d00:     0x00000001      0x00000002      0xbfed9d28      0x080483e6
0xbfed9d10:     0x0075eff4      0x0075eff4      0x080494c4      0x0075eff4
0xbfed9d20:     0x00000002      0x00000001
(gdb) disas main
Dump of assembler code for function main:
0x0804838e <main+0>:    push   %ebp
0x0804838f <main+1>:    mov    %esp,%ebp
0x08048391 <main+3>:    sub    $0x8,%esp
0x08048394 <main+6>:    and    $0xfffffff0,%esp
0x08048397 <main+9>:    mov    $0x0,%eax
0x0804839c <main+14>:   add    $0xf,%eax
0x0804839f <main+17>:   add    $0xf,%eax
0x080483a2 <main+20>:   shr    $0x4,%eax
0x080483a5 <main+23>:   shl    $0x4,%eax
0x080483a8 <main+26>:   sub    %eax,%esp
0x080483aa <main+28>:   movl   $0x1,0xfffffffc(%ebp)
0x080483b1 <main+35>:   movl   $0x2,0xfffffff8(%ebp)
0x080483b8 <main+42>:   sub    $0x8,%esp
0x080483bb <main+45>:   pushl  0xfffffff8(%ebp)
0x080483be <main+48>:   pushl  0xfffffffc(%ebp)
0x080483c1 <main+51>:   call   0x8048368 <fn>
0x080483c6 <main+56>:   add    $0x10,%esp
0x080483c9 <main+59>:   leave
0x080483ca <main+60>:   ret
End of assembler dump.
(gdb)
```

return address

checks out – is the right resumption location to pick up where we left off

Return address location formula: $ebp+4 *

```
(gdb) print $esp
$3 = (void *) 0xbfed9cf0
(gdb) print $ebp
$4 = (void *) 0xbfed9cf8
(gdb) x/14 $esp
0xbfed9cf0:     0x00000004      0x00000003      0xbfed9d28      0x080483c6
0xbfed9d00:     0x00000001      0x00000002      0xbfed9d28      0x080483e6
0xbfed9d10:     0x0075eff4      0x0075eff4      0x080494c4      0x0075eff4
0xbfed9d20:     0x00000002      0x00000001
(gdb) disas main
Dump of assembler code for function main:
0x0804838e <main+0>:    push   %ebp
0x0804838f <main+1>:    mov    %esp,%ebp
0x08048391 <main+3>:    sub    $0x8,%esp
0x08048394 <main+6>:    and    $0xfffffff0,%esp
0x08048397 <main+9>:    mov    $0x0,%eax
0x0804839c <main+14>:   add    $0xf,%eax
0x0804839f <main+17>:   add    $0xf,%eax
0x080483a2 <main+20>:   shr    $0x4,%eax
0x080483a5 <main+23>:   shl    $0x4,%eax
0x080483a8 <main+26>:   sub    %eax,%esp
0x080483aa <main+28>:   movl   $0x1,0xfffffffc(%ebp)
0x080483b1 <main+35>:   movl   $0x2,0xfffffff8(%ebp)
0x080483b8 <main+42>:   sub    $0x8,%esp
0x080483bb <main+45>:   pushl  0xfffffff8(%ebp)
0x080483be <main+48>:   pushl  0xfffffffc(%ebp)
0x080483c1 <main+51>:   call   0x8048368 <fn>
0x080483c6 <main+56>:   add    $0x10,%esp
0x080483c9 <main+59>:   leave
0x080483ca <main+60>:   ret
End of assembler dump.
(gdb)
```

return address

+4=0xbfed9cfc

checks out – is the right resumption location to pick up where we left off

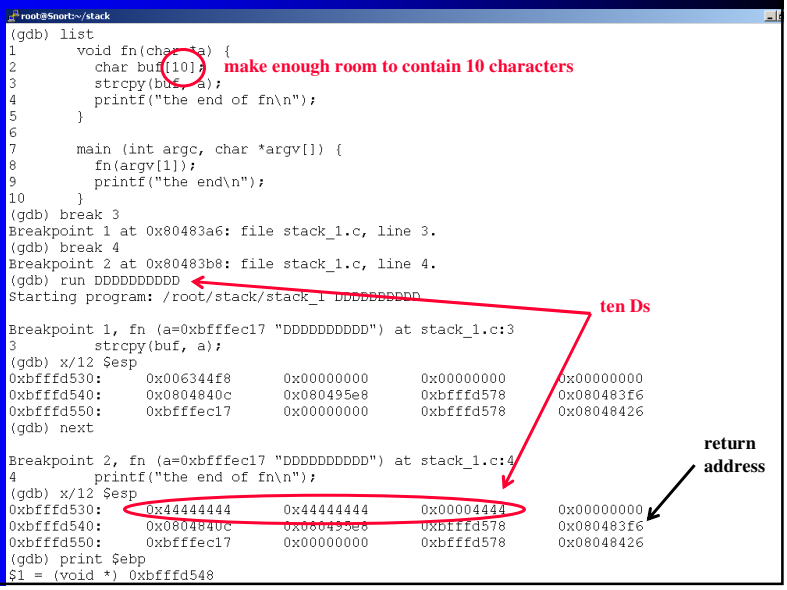
*just in case you ever want to overwrite it

stack_1.c – fixed param space, but variable arg len*

```
void fn(char *a) {
  char buf[10];
  strcpy(buf, a);
  printf("the end of fn\n");
}

main (int argc, char *argv[]) {
  fn(argv[1]);
  printf("the end\n");
}
```

*parameter - placeholder variable in function definition for receiving a passed value
argument – specific value that is passed

Stack separation between argument & return address



Crafting an attack based on this

- control argument length
  - extend enough to overwrite the return address
- control argument content
  - craft meaningful code into early portion
  - calculate overwritten return address value to backpoint into that code

How?

- this exercise ends with article's page 8
- keep reading, page 9 (extracurricular)…
  - gives a real-world example
  - delivers malicious argument across a network
  - achieves a shell prompt

Please see

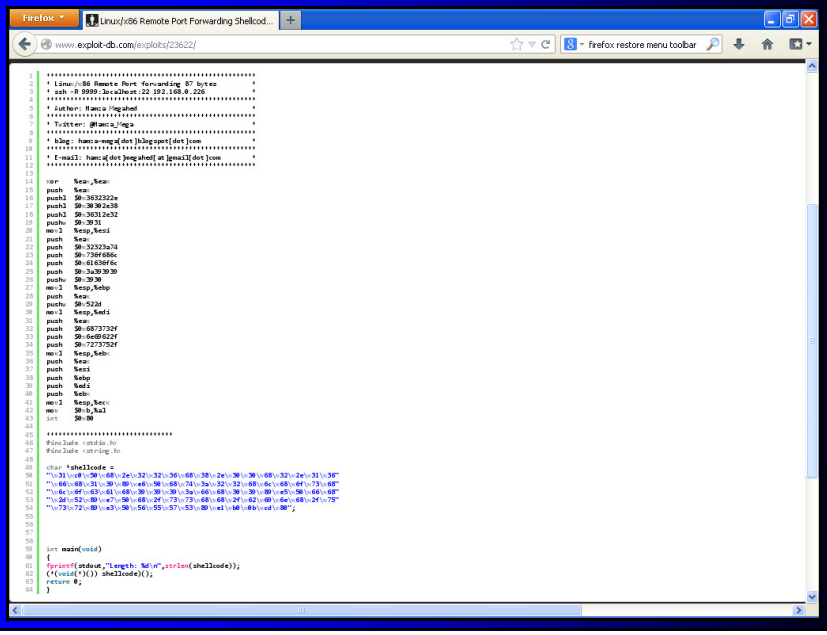
- "Overflowing the stack on Linux x/86"
  - http://www-scf.usc.edu/~csci530l/downloads/stackoverflow_en.pdf
  - originally http://sobolewscy.in5.pl/piotr/publikacje/hakin9/stackoverflow_en.pdf
- GNU debugger (gdb) documentation
  - http://sourceware.org/gdb/current/onlinedocs/gdb.html#SEC_Top

Any other code suffer this feature?

if we knew
about it, no
(it'd be fixed by now)

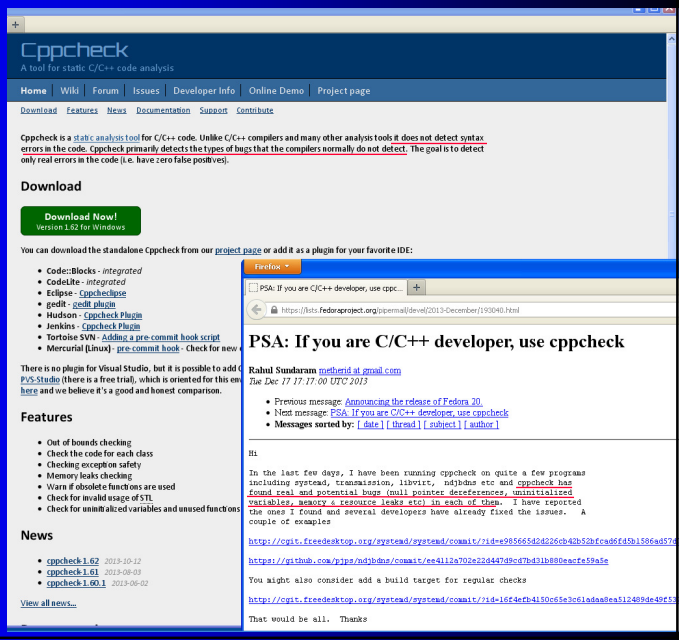
but we don't,
Yes (lots)

Here's one



What can be done?

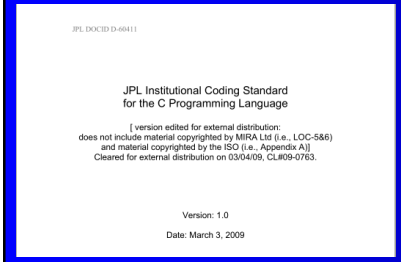
- tighten compiler checks
  - this lab might not work with later gcc releases
- perform static code analysis



A 2014 "current" event

security system needed in space??

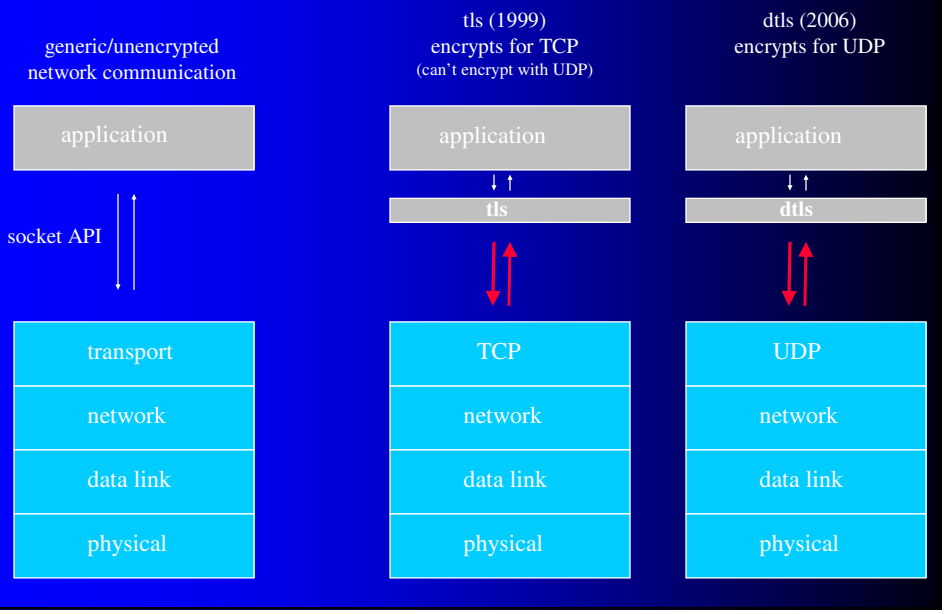
9/25/15



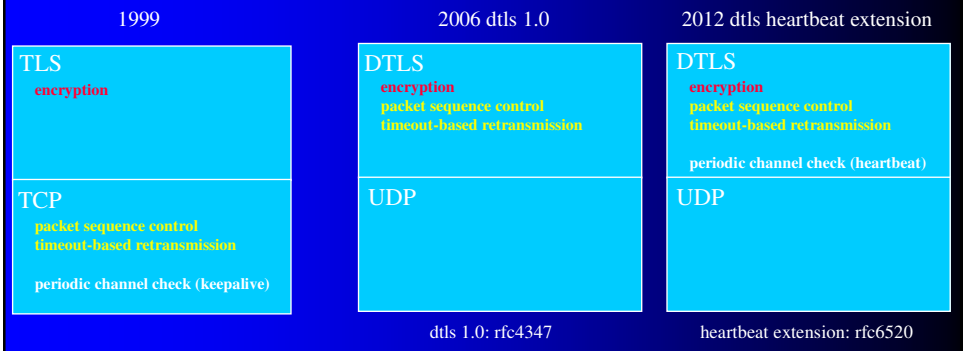
"For instance, an area of memory above the stack limit allocated to each task should be reserved as a safety margin, and filled with a fixed and uncommon bit-pattern. A health task can detect stack overflow anomalies by at regular intervals checking the presence of the bit-pattern for each task. The same principle can be used to protect against buffer overflow, or access to memory outside allocated regions. Critical parameters should similarly be protected in memory by placing safety margins and barrier patterns around them, so that access violations and data corruption can be detected more easily."

heartbleed bounds checking oversight

Encrypting: for TCP vs for UDP



Distribution of function between protocol layers



dtls 1.0: rfc4347

heartbeat extension: rfc6520

packet ordering essential for tls/dtls encryption
- tls gets it from tcp
- dtls must provide it (because udp does not)

channel check nonessential, but nice
- tls gets it from tcp as "keepalive"
- dtls added it as "hearbeat"

Heartbeat extension rfc6520

```
Internet Engineering Task Force (IETF)                      R. Seggelmann
Request for Comments: 6520                                      M. Tuexen
Category: Standards Track                 Muenster Univ. of Appl. Sciences
ISSN: 2070-1721                                                M. Williams
                                                      GWhiz Arts & Sciences
                                                             February 2012


                  Transport Layer Security (TLS) and
      Datagram Transport Layer Security (DTLS) Heartbeat Extension

Abstract

   This document describes the Heartbeat Extension for the Transport
   Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
   protocols.

   The Heartbeat Extension provides a new protocol for TLS/DTLS allowing
   the usage of keep-alive functionality without performing a
   renegotiation and a basis for path MTU (PMTU) discovery for DTLS.
```

rfc6520 excerpts

"…The Heartbeat protocol is a new protocol running on top of the Record Layer [of ssl]. The protocol itself consists of two message types: HeartbeatRequest and HeartbeatResponse….

"The Heartbeat protocol messages consist of their type and an arbitrary payload and padding.

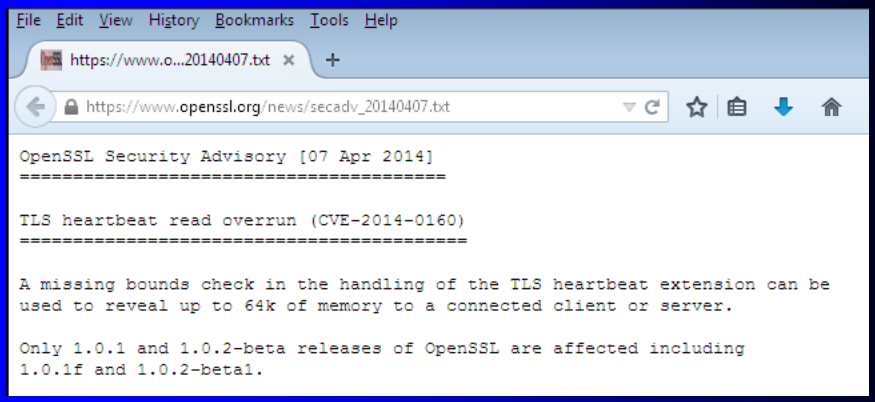
```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
```

"…payload: The payload consists of arbitrary content.

"…If the payload_length of a received HeartbeatMessage is too large, the received HeartbeatMessage MUST be discarded silently.

"When a HeartbeatRequest message is received … the receiver MUST send a corresponding HeartbeatResponse message carrying an exact copy of the payload of the received HeartbeatRequest…. "

Breaking news…



The effect

Heartbeat sent to victim

SSLv3 record:

Length
4 bytes

HeartbeatMessage:

Type	Length	Payload data
TLS1_HB_REQUEST	65535 bytes	1 byte

Victim's response

SSLv3 record:

Length
65538 bytes

HeartbeatMessage:

Type	Length	Payload data	
TLS1_HB_RESPONSE	65535 bytes	65535 bytes	

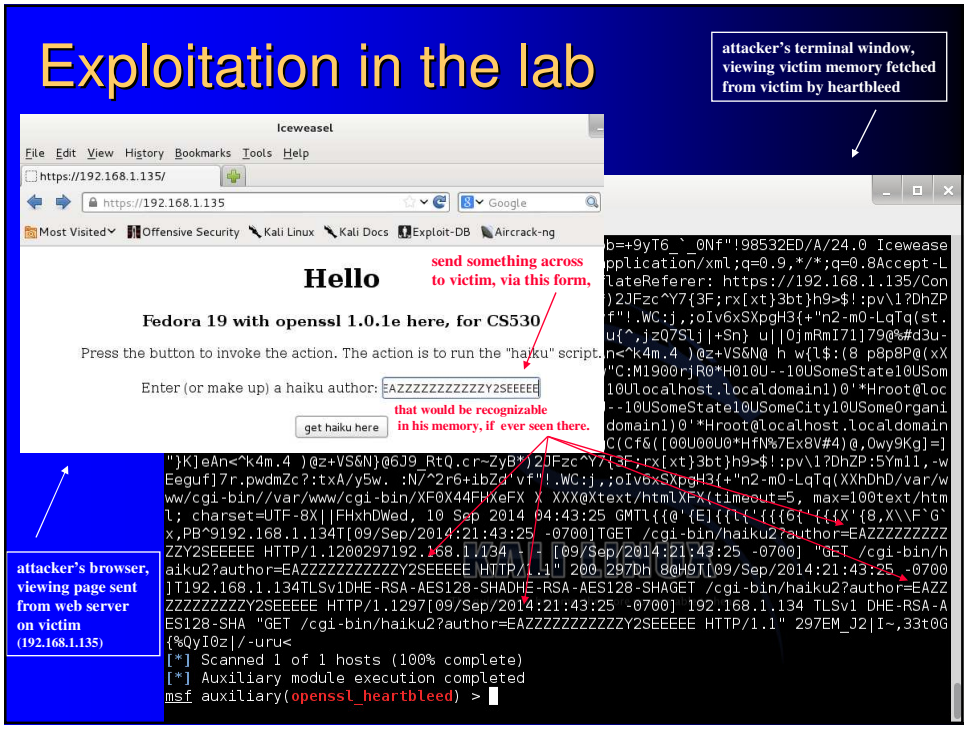
http://www.theregister.co.uk/2014/04/09/heartbleed_explained/ see also: https://xkcd.com/1354/

The fix

http://pastebin.com/5PP8JVqA

```
$ diff -r openssl-1.0.1f/ssl/t1_lib.c openssl-1.0.1g/ssl/t1_lib.c
#ifndef OPENSSL_NO_HEARTBEATS                   #ifndef OPENSSL_NO_HEARTBEATS
int                                             int
tls1_process_heartbeat(SSL *s)                  tls1_process_heartbeat(SSL *s)
        {                                               {
        unsigned char *p = &s->s3->rrec.data[0], *pl;           unsigned char *p = &s->s3->rrec.data[0], *pl;
        unsigned short hbtype;                          unsigned short hbtype;
        unsigned int payload;                           unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */        unsigned int padding = 16; /* Use minimum padding */

        /* Read type and payload length first */
        hbtype = *p++;
        n2s(p, payload);
        pl = p;

        if (s->msg_callback)                            if (s->msg_callback)
                s->msg_callback(0, s->version, TLS1_RT_HEARTB           s->msg_callback(0, s->version, TLS1_RT_HEARTB
                        &s->s3->rrec.data[0], s->s3->rrec.len                   &s->s3->rrec.data[0], s->s3->rrec.len
                        s, s->msg_callback_arg);                                s, s->msg_callback_arg);

                                                        /* Read type and payload length first */
                                                        if (1 + 2 + 16 > s->s3->rrec.length)
                                                                return 0; /* silently discard */
                                                        hbtype = *p++;
                                                        n2s(p, payload);
                                                        if (1 + 2 + payload + 16 > s->s3->rrec.length)
                                                                return 0; /* silently discard per RFC 6520 se
                                                        pl = p;

        if (hbtype == TLS1_HB_REQUEST)                  if (hbtype == TLS1_HB_REQUEST)
```

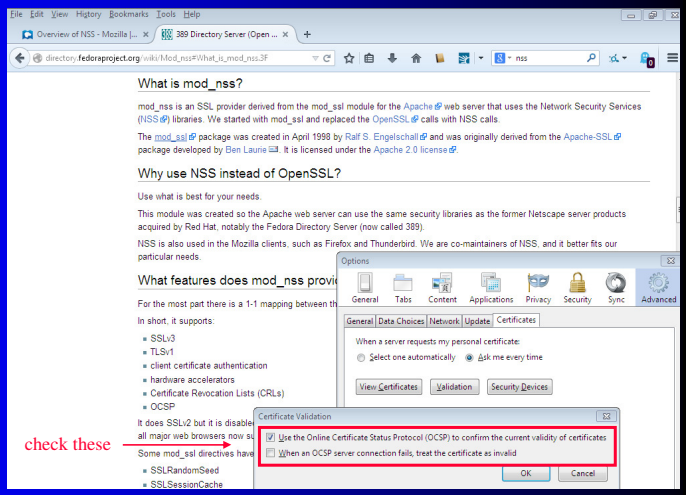
Exploitation in the lab



Don't let browser accept revoked certs

require affirmative non-rev check

- server sites remediate by 1-updateing OpenSSL 2-revoking certificates (to prevent site impersonation via possible previous heartbleed-exfiltrated private keys)
- only meaningful if client (you!) does his part, i.e., checks for the revocation and honors it
- turn it on in your browser if it supports it
- Firefox does; phones' browsers probably don't



http://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html

My faq

Q. Is this an exploitation of the SSL/TLS protocol?
A. No, it's an exploitation of the OpenSSL implementation of it.

Q. Are there other implementations?
A. Yes for example Mozilla's NSS (Network Security Services) or GnuTLS

Q. How widespread among websites is the use of OpenSSL to provide TLS?
A. Maybe 17.5% of them use OpenSSL for that
http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html

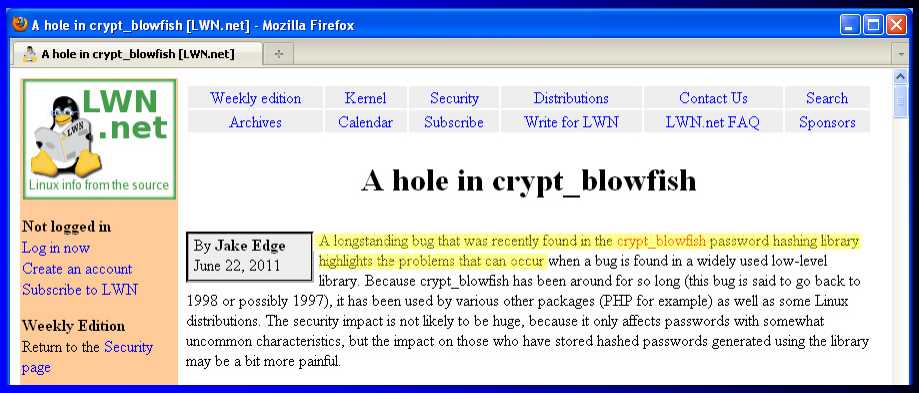
Q. Does Apache use OpenSSL for SSL?
A. Yes, if it uses mod_ssl for ssl. But it could use mod_nss and thus NSS's ssl. Usually it installs with mod_ssl by default.
http://directory.fedoraproject.org/wiki/Mod_nss#What_is_mod_nss.3F

Information sources

- code analyses:
  http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html
  http://www.theregister.co.uk/2014/04/09/heartbleed_explained/
- Security Now podcast - "How the Heartbleeds"
  - audio: https://media.grc.com/sn/sn-450-lq.mp3
  - transcript: https://www.grc.com/sn/sn-450.pdf
  - shownotes: https://www.grc.com/sn/sn-450-notes.pdf
- http://heartbleed.com

C language sign extension bug

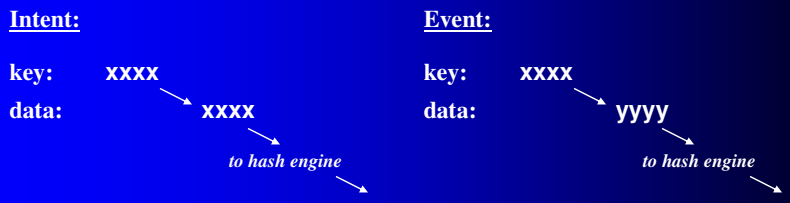
Case study - a longstanding bug



- introduced late 90s, noticed then but overlooked ever since
- rediscovered while testing John the Ripper in June 2011
- in the crypt_blowfish library
- freely, admirably, immediately admitted, documented, and fixed by the library's author (who is also author of John the Ripper)

What was the bug?

- 4 bytes of key/password needed to be hashed
  - passed to a char-type parameter variable "key"
  - transferred to long(4-byte)-type variable "data"
- the transfer went bad
  - "data" ended with value *different* from "key"
- resulting hash *not* that of the password



Underlying background issues

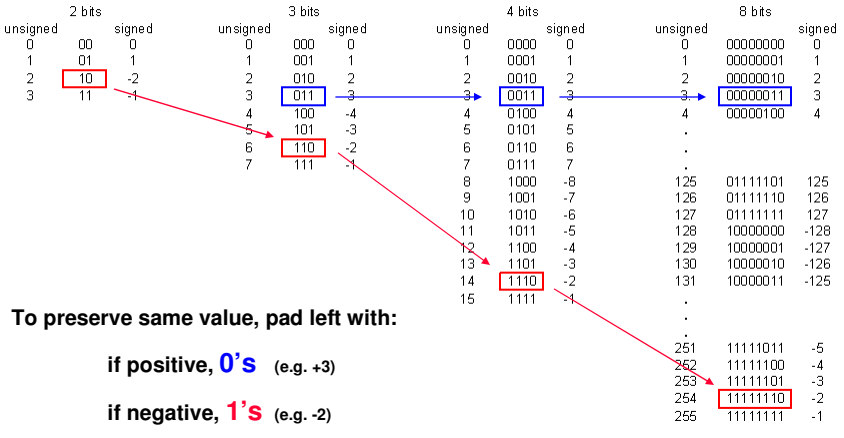
- binary signed integer representation
- the bitwise OR operation

Representing signed integers
(two's complement method)

Split range in half
- low value half for zero and positive
- high value half for negative

	value unsigned		value signed	
0000	0	0000	0	
0001	1	0001	1	
0010	2	0010	2	zero and positive
0011	3	0011	3	
0100	4	0100	4	
0101	5	0101	5	
0110	6	0110	6	
0111	7	0111	7	
1000	8	1000	-8	
1001	9	1001	-7	
1010	10	1010	-6	
1011	11	1011	-5	negative
1100	12	1100	-4	
1101	13	1101	-3	
1110	14	1110	-2	
1111	15	1111	-1	

Widening signed integers
"extension" and "sign extension"



To preserve same value, pad left with:

if positive, 0's (e.g. +3)

if negative, 1's (e.g. -2)

Background: OR operation

- an operation
  - operands (input): 2 bits
  - result (output): 1bit
- ORing a bit with 0 yields (preserves) that bit

  0 OR **0** = 0

  1 OR **0** = 1
- ORing a bit with 1 yields 1 unconditionally

  0 OR **1** = 1

  1 OR **1** = 1

ORing 2 bytes with each other

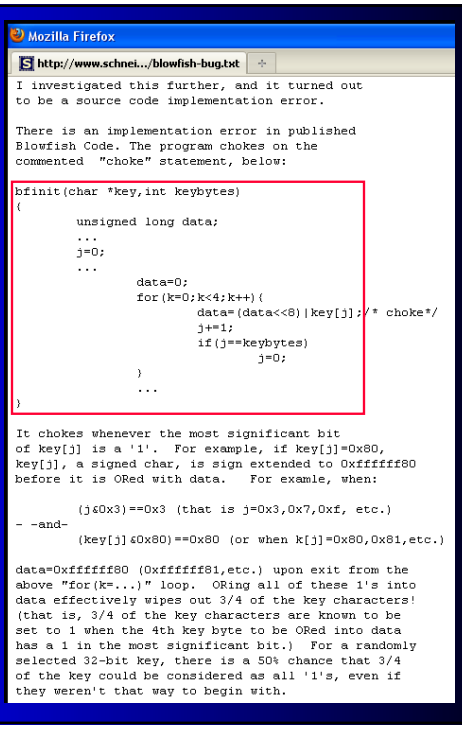
- no such thing
  - OR is an operation for pairs of bits only
  - not pairs of aces, nor deuces, nor bytes
- "ORing bytes" signifies 8 normal (bitwise) ORs, collectively

"ORing 2 bytes" == 8 of these

ORing words requires 2 words, of equal length, to enable ORing their bits

```
 01001001
 10011010
_10011011
```

The offending code



offending code
loads key's 4 bytes into data

	key[0]	key[1]	key[2]	key[3]
key:	00010001	00100010	01000100	10001000
data:	10101010	10111011	11001100	11011101

initial value is random/garbage

by doing this:

data=(data<<8) | key[j]

4 times

data=(data<<8) | key[j]

Observation

- data is 4 bytes wide
- key[j] is only 1 byte
- key[j] is *too short* to OR with data
- so pad ("extend") it by 24 bits on the left

Operation

1. shift 'data' 8 bits left
   - left byte disappears
   - right byte zero-filled
2. left-pad key[j] with 24 zeros
3. OR them together
   - extended key[j]'s zeros preserve data's leftmost 3 bytes
   - data's zeros preserve extended key[j]'s rightmost byte
4. assign result to data

Intended operation of algorithm

initial "key" 00010001	00100010	01000100	10001000
initial "data" 10101010	10111011	11001100	11011101

evolution of"data":

iteration 1	shift	10111011 11001100 11011101 *00000000*
	extend	00000000 00000000 00000000 00010001
	or	10111011 11001100 11011101 00010001
iteration 2	shift	11001100 11011101 00010001 *00000000*
	extend	00000000 00000000 00000000 00100010
	or	11001100 11011101 00010001 00100010
iteration 3	shift	11011101 00010001 00100010 *00000000*
	extend	00000000 00000000 00000000 01000100
	or	11011101 00010001 00100010 01000100
iteration 4	shift	00010001 00100010 01000100 *00000000*
	extend	00000000 00000000 00000000 10001000
	or	00010001 00100010 01000100 10001000

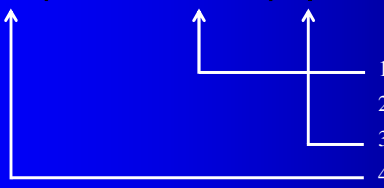
00000's from extend

00000's from shift

final "data" holds initial "key" ⟶

step 2: implicit, *lexically invisible*

```
data=(data<<8) | key[j]
```



1. shift 'data' 8 bits left
2. left-pad key[j] with 24 zeros
3. OR them together
4. assign result to data

Actual operation of algorithm

initial "key" 00010001	00100010	01000100	10001000
initial "data" 10101010	10111011	11001100	11011101

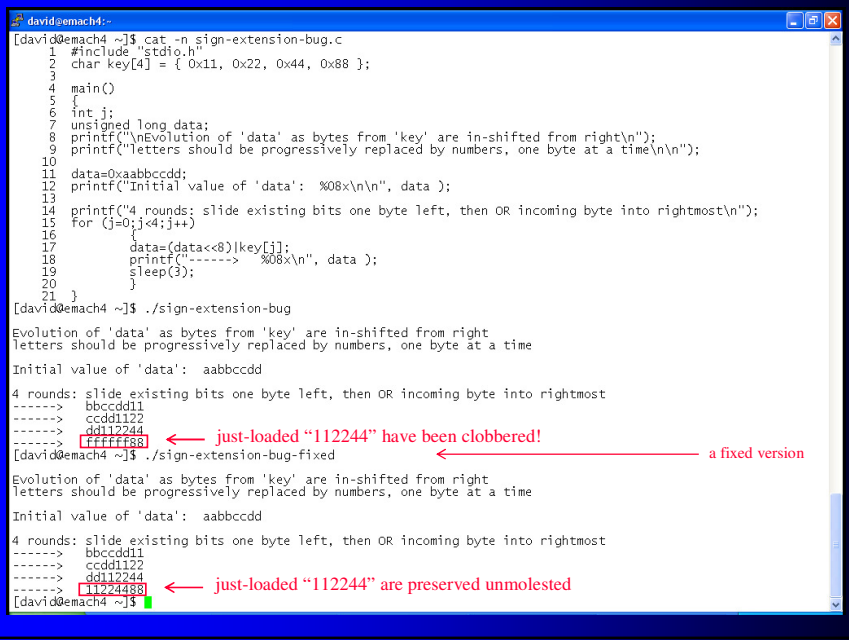
evolution of"data":

iteration 1	shift	10111011 11001100 11011101 *00000000*
	extend	00000000 00000000 00000000 00010001
	or	10111011 11001100 11011101 00010001
iteration 2	shift	11001100 11011101 00010001 *00000000*
	extend	00000000 00000000 00000000 00100010
	or	11001100 11011101 00010001 00100010
iteration 3	shift	11011101 00010001 00100010 *00000000*
	extend	00000000 00000000 00000000 01000100
	or	11011101 00010001 00100010 01000100
iteration 4	shift	00010001 00100010 01000100 *00000000*
	extend	11111111 11111111 11111111 10001000
	or	11111111 11111111 11111111 10001000

problem arises here

final "data" does ***not*** hold initial "key" ⟶

A code embodiment



Why is this happening?

- because C by default treats char type as signed
- So hex 88 (= bin 10001000) treated is as if
  - decimal -120
  - not decimal 136
- extend from 1 to 4 bytes keeping -120 value needs
  - left-pad with 1
  - not left-pad with 0
- alters the subsequent OR operation

What effects?

- replaces many password characters with FF
- promotes FF to ranks of high predictability
  - along with natural language words
  - along with birthday strings
  - along with pets' names
- eases intelligent brute force cracking task
  - FF-heavy guesses are now rewarding to try a lot

What effects?

"I am wondering ... why I am getting different hashes...."

"...it means we have incorrect (incompatible with OpenBSD's) hashes in the wild..."

"John the Ripper and crypt_blowfish developer Alexander Peslyak (aka Solar Designer) analyzed the effects of the bug and found that some password pairs would hash to the same value with only minimal differences (e.g. "ab£" hashed to the same value as "£"), which would make password cracking easier. A further analysis shows that some characters appearing just before one with the high bit set may be effectively ignored when calculating the hash. That would mean that a simpler password than that given by the user could be used and would still be considered valid—a significant weakening of the user's password.

"It should be noted that Solar Designer has been very forthcoming with details of the problem and its effects."

See: **http://lwn.net/Articles/448699/**
http://lwn.net/Articles/448723/
http://lwn.net/Articles/448725/

Observations

- a C-language-specific problem
- assembler would be immune
  - left-pad/extension is lexically explicit/inescapable
  - MOVZX, "move zero extend" – use 0s, versus MOVSX, "move sign extend" – use 1s
- will not affect ascii password characters
  - they fall in the "positive" range of signed representation
  - none have the offending, triggering leading 1-bit
  - but not all passwords/keys are human generated ascii

Information sources

- http://lwn.net/Articles/448699/
- http://lwn.net/Articles/448723/
- http://lwn.net/Articles/448725/
- http://www.schneier.com/blowfish-bug.txt
- Security Now podcast - "Anatomy of a Security Mistake"
  - audio: http://media.grc.com/SN/sn-311-lq.mp3
  - transcript: http://www.grc.com/sn/sn-311.txt