Memory Corruption
Buffer Overflows

Agenda

- Overview of buffer overflows
  - Stack-based
  - Structured Exception Handlers (SEH)
  - Heap-based
- Buffer overflow myths
- Reducing the risk of buffer overflow attacks in code with the Microsoft SDL
- Common Weakness Enumeration (CWE) Overview
- Examples
- Conclusions

Buffer Overflows Overview

Buffer Overflow: Occurs when data is written into a fixed-length buffer and the size of that data exceeds the capacity of the receiving buffer
A more general definition: any access (R/W) outside the reserved area (under/over)

- **Primary Risks:** Corrupt data, crash programs and control execution flow
- Common in native applications (C/C++)
  - Rare, but still possible in managed code (.NET, Java)
- Cause is failing to validate input
- Can occur on stacks and heaps

Context

- process memory layout
  - code: program code and libraries
  - data: global and static variables, also the heap
  - stack: function’s arguments and local variables, control data (e.g. return address)
- data, control information and code mixed together
  - code / control information could be overwritten
  - system confuse data with code

The Classical Unsafe strcpy() Example

```
char dst[5];
char *src = "0123456789";
strcpy(src, dst);
```

Review of Application Stack Frames

```
void main(void)
{
    FunctionOne(arguments);
    FunctionTwo();
}
```

```
void FunctionOne(int c)
{
    /* Operations */
}
```

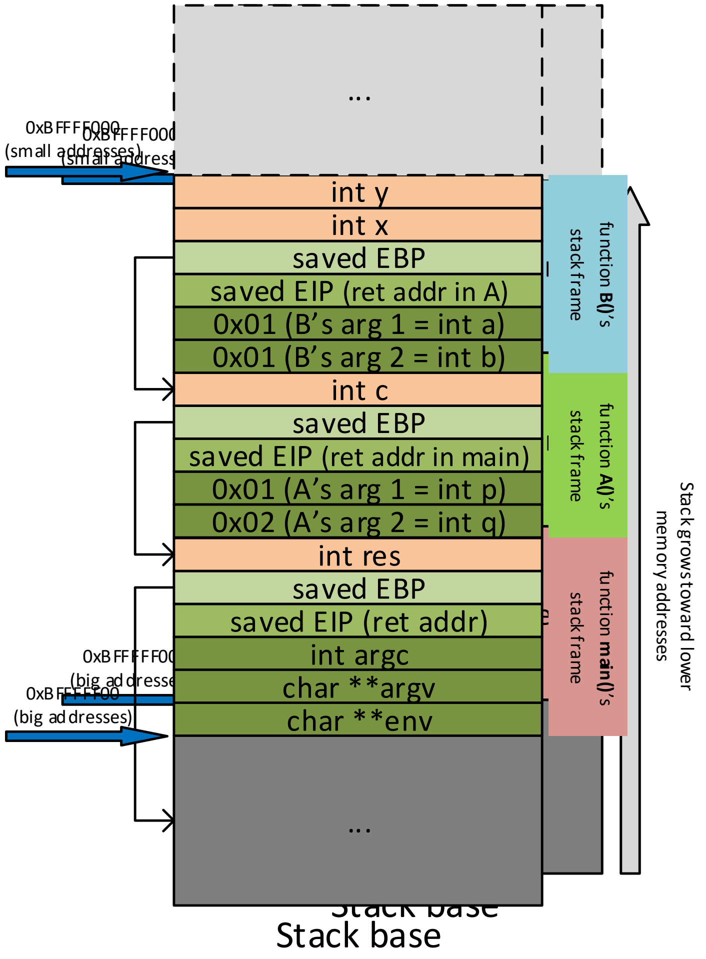
```
void FunctionTwo(void)
{
    int LocalInt;
    char LocalBuffer[32];

    /* Operations */
}
```

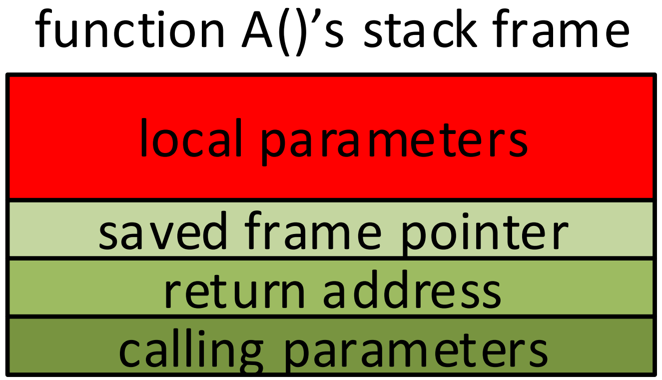
Local function variables	Saved Frame Pointer	Return Address	Function parameters

Review of Application Stack Frames (detailed)

```
int function_B(int a, int b)
{
  int x, y; // local variable
  x = a * a;
  y = b * b;

  return (x + y);
}

int function_A(int p, int q)
{
  int c; // local variables
  c = p * q * function_B(p, p);

  return c;
}

int main(int argc, char **argv, char **env)
{
  int res;
  res = function_A(1, 2);
  ...
  return res;
}
```

```
  push EBP
  mov  EBP, ESP
  sub  ESP, 48h

  push EBP
  mov  EBP, ESP
  sub  ESP, 44h

  push 1
  push 1
  call function_B

  push  2
  push  1
  call function_A
```





Reference. M. Down et al., *The Art of Software Security Assessment*, Addison Wesley, 2012, pg. 175

Exploitation: Overwrite Function's Local Variable or Arguments

- function's calling arguments and local variables are on the stack
  - the overflowed buffer among them
  - ⇒ could overwrite nearby locations
- very application specific
- depends on the way the compiler generates code
  - calling convention, stack organization

Exploitation: Overwrite Function's Local Variable or Arguments (cont.)

- example: local variable authenticated compromised ⇒ application functionality altered

```
int authenticate(char *username, char *password)
{
  int authenticated;
  char buffer[1024];

  authenticated = verify_password(username, password);

  if (authenticated == 0) {
    sprintf(buffer, "password is incorrect for user %s\n", username);
    log("%s", buffer);
  }

  return autheticated;
}
```

Exploitation: Overwrite Control Data

- overwrite the return address ⇒ execution could be returned
- to an area of memory containing data the attacker controls
  - ex.: global variables, a stack location, a static buffer filled with attacker's code
  - attackers injected code: shellcode (tries opening a remotely accessible shell)
  - based on confusion between data and code
  - possible if injected code could be executed
- somewhere into the application code or in a shared library
  - where some code useful for the attacker is located
  - e.g. the call to a system function in a library
  - independent on attacker's injected code

Stack-Based Buffer Overflows

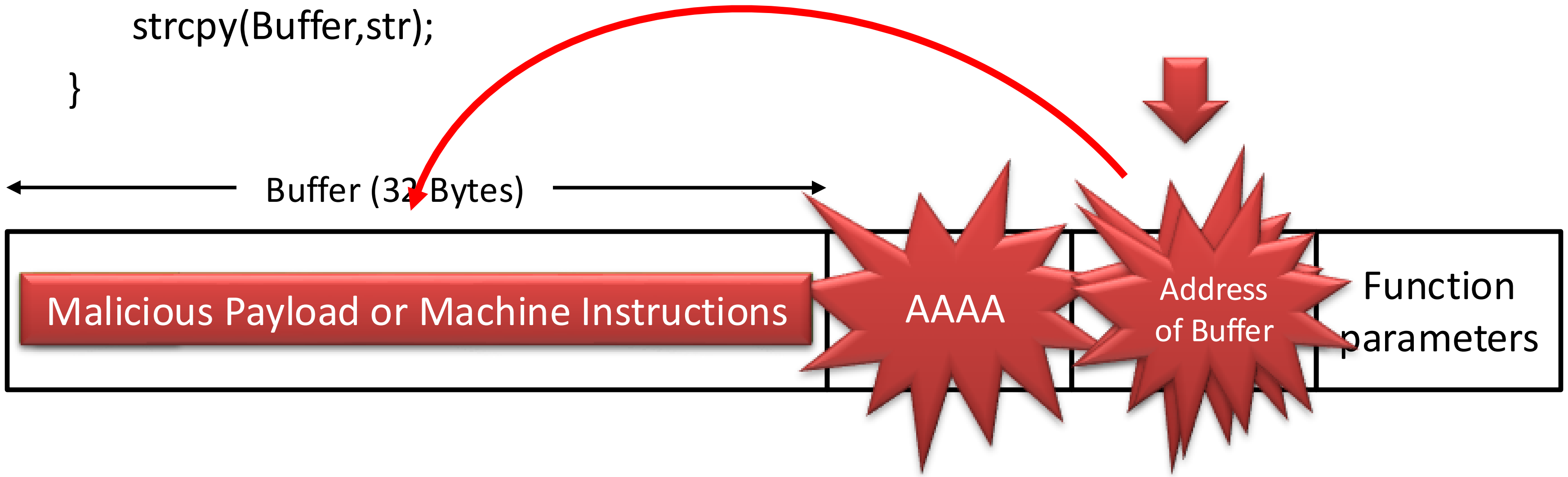
Primary Risk: Ability to overwrite control structures

/* UNSAFE Function */

```
void UnsafeFunction(char * str)
{
    char Buffer[32];

    /* Copy str into Buffer */
    strcpy(Buffer,str);
}
```

SAMPLE INPUTS (STR VALUES):

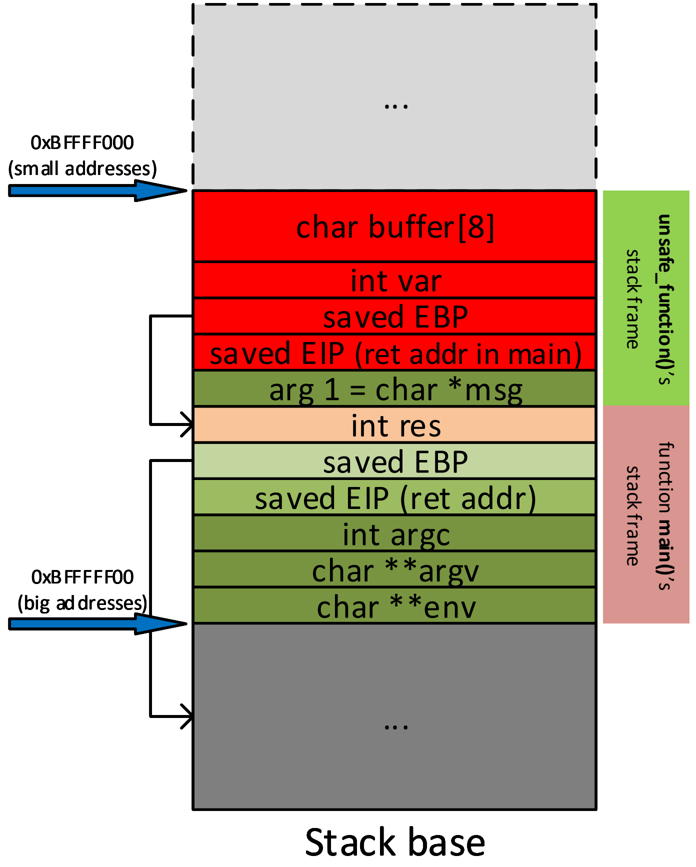
1. "Kevin"
2. "A" repeated 40 times



Stack-Based Buffer Overflows (details)

Primary Risk: Ability to overwrite control structures

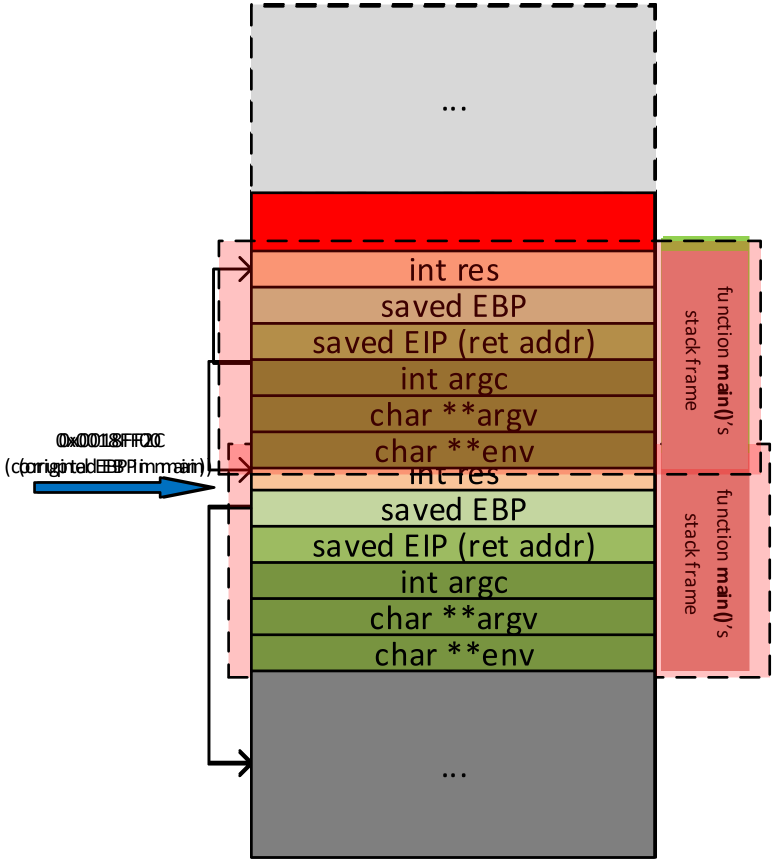
```
int unsafe_function(char *msg)
{
  int var;           // local variables
  char buffer[8];

  var = 10;
  strcpy(buffer, msg);

  return var;
}

int main(int argc, char **argv, char **env)
{
  int res;

  /* Buffer overflow for "strlen(argv[1]) >= 8"
  res = unsafe_function(argv[1]);

  return res;
}
```



Off-by-One Stack-Based Buffer Overflows

Primary Risk: Ability to overwrite local variables or saved EBP

```
int unsafe_function(char *msg)
{
  char buffer[512]; // local variables

  // wrong limit checking
  if (strlen(msg) <= 512)
    strcpy(buffer, msg);
}

int main(int argc, char **argv, char **env)
{
  int res;

  /* Buffer overflow for "strlen(argv[1]) >= 8"
  res = unsafe_function(argv[1]);

  return res;
}
```



Stack base

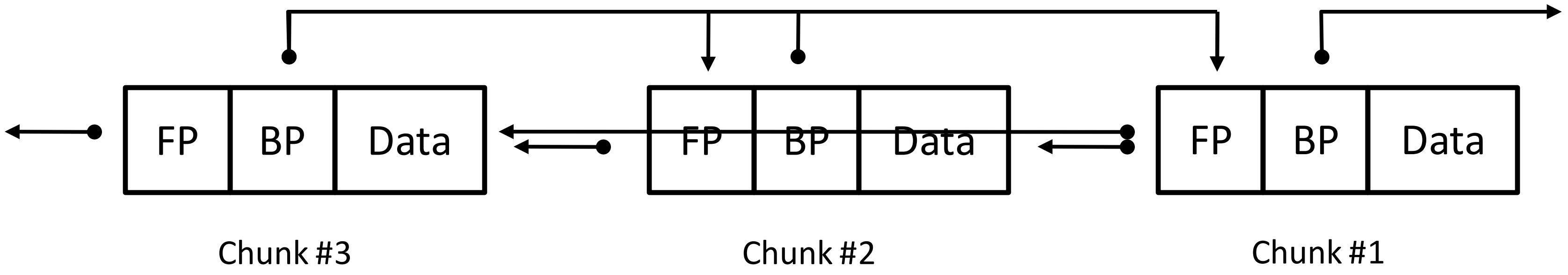
Review of Application Heaps

```
void SampleFunction(void)
{
        /* Allocate space on heap */
        char * ptr = (char *)malloc(32);

        /* Operations */

        /* Free allocated heap space */
        free(ptr);
}
```

Pseudo-code For Chunk Freeing:

NextChunk = Current->FP
PreviousChunk = Current->BP

NextChunk->BP = PreviousChunk
PreviousChunk->FP = NextChunk



Heap-Based Buffer Overflows

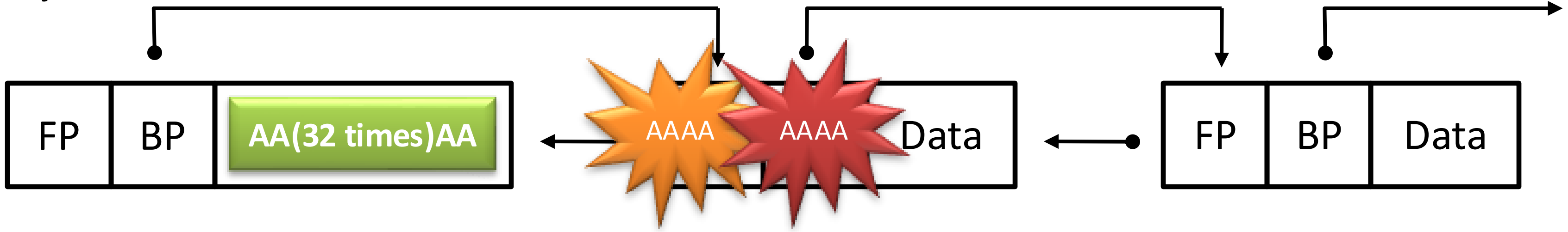
Primary Risk: Ability to write arbitrary 4 byte DWORD anywhere in memory (return address, pointers, etc.)

```
/* UNSAFE Function */
void UnsafeFunction(char * str)
{
    /* Allocate 32 bytes heap space */
    char * Buffer = (char *)malloc(32);

    /* Copy str into Buffer */
    strcpy(Buffer,str);
}
```

Pseudo-code For Chunk Freeing:

NextChunk = AAAA
PreviousChunk = AAAA

AAAA ->BP = AAAA
PreviousChunk->FP = NextChunk



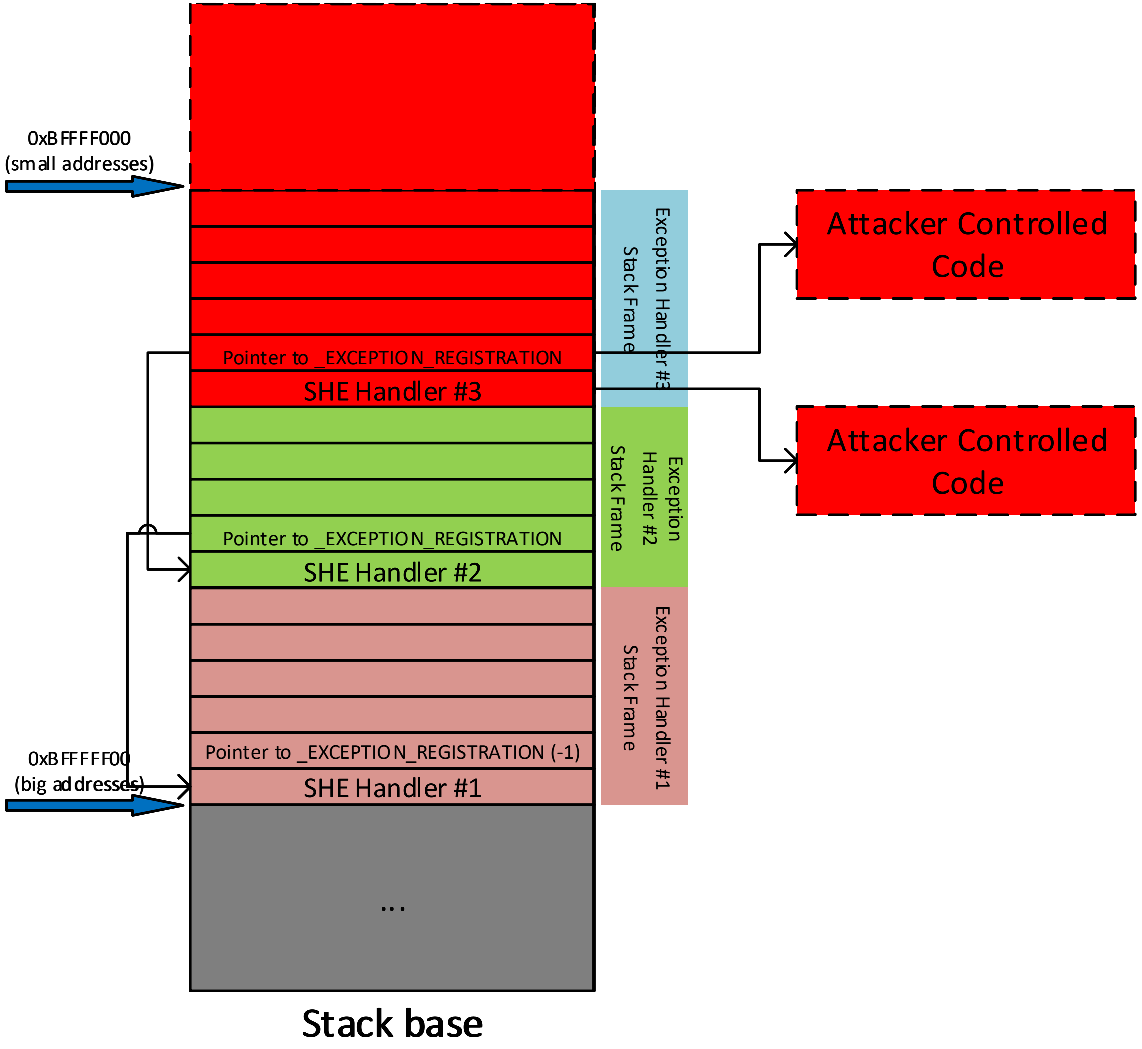
Chunk #3 Chunk #2 Chunk #1

Structured Exception Handling (SEH)

- specific to Windows
- programs could register handlers to act on errors
  - catching exceptions thrown by the program during runtime
- exception handler registration structures are located on the stack and contains
  - address of a handler routine
  - pointer to its parent handlers
- the exception handler chain is traversed from the most recently installed handler back to the first one
  - identify the appropriate handler, by executing each one in turn
- if an attacker could perform stack overflow
  - could overwrite the exception handling structure
  - than generate an exception
  - the execution could jump to the attacker's controlled address

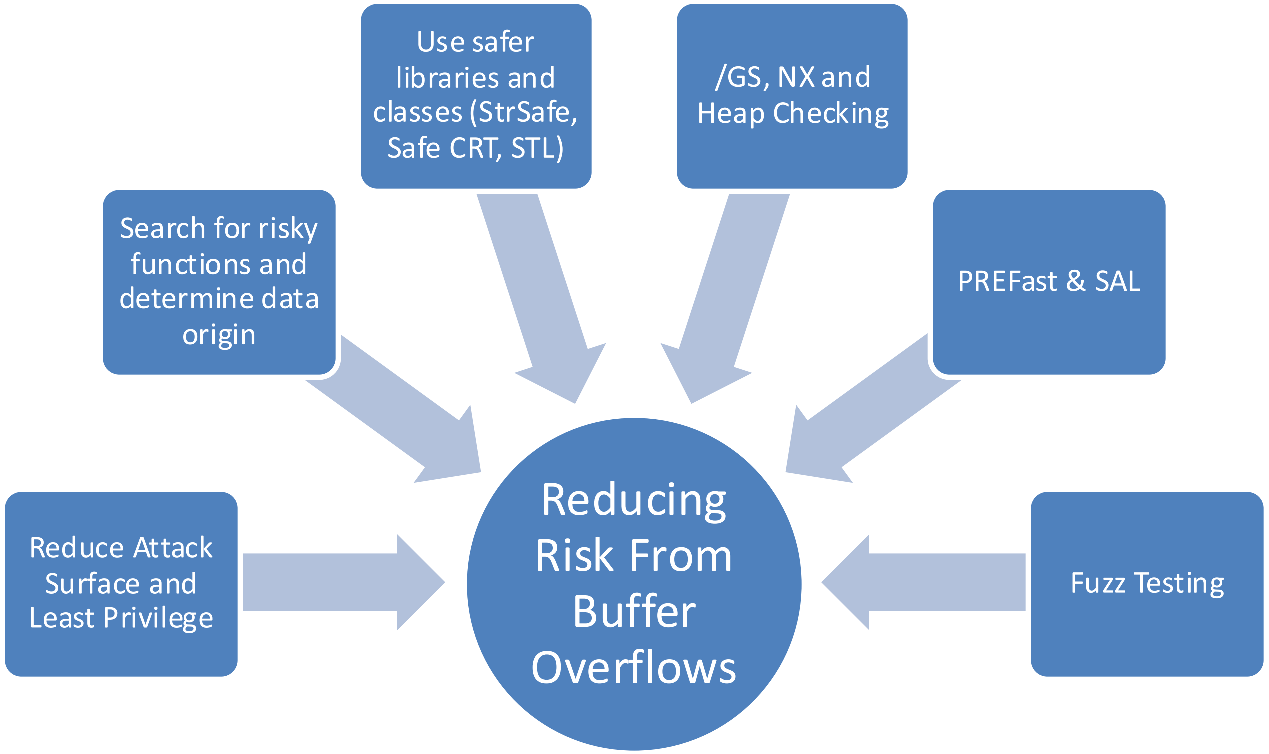
Reference. M. Down et al., *The Art of Software Security Assessment*, Addison Wesley, 2012, pg. 179-180

Structured Exception Handling (SEH)



Reference. M. Down et al., *The Art of Software Security Assessment*, Addison Wesley, 2012, pg. 179-180

Reducing Exposure to Buffer Overflows with the Microsoft SDL



- *Presentation content is available for all of these topics*

SDL: Review Source Code for Buffer Overflows

- **Source code review:** Manual inspection of application for specific vulnerabilities, such as buffer overflows
  - Input received from network, file, command line
  - Transfer of received input to internal structures
  - Use of unsafe string handling calls
  - Use of arithmetic to calculate an allocation or remaining buffer size
- Overall method: trace user input from the entry point of the application through all function calls

Reference. M. Howard et al., "24 Deadly Sins of Software Security", 2010, p. 99-100

SDL: Use Safer APIs and Avoid Banned APIs

- **Safer APIs:** Development libraries that are more resistant to buffer overflows
- **Banned APIs:** Development libraries that can easily lead to buffer overflows, and banned for use by the Microsoft SDL

SDL:
Use Run-Time Protection

- **Compiler Protection:** Run-time checks that reduce risk from buffer overflow attacks

SDL:
Use Code Analysis Tools

- **Code Analysis Tools:** Automated tools designed to aid in the identification of known vulnerabilities in code

SDL:
Use Fuzz Testing

- **Fuzz Testing:** A testing methodology that can help identify security issues that manifest in applications due to improper input validation

Platform Protection From Buffer Overflows

- Modern day operating systems and processors have built-in buffer overflow protection
  - Address Space Layout Randomization (ASLR)
  - Data Execution Protection (DEP)
- However none of these are “silver bullets”
  - Denial of Service (DoS) attacks usually not prevented
  - More subtle attacks could still be performed
  - Developers still need to follow security best practices
  - Developers should always apply the Microsoft SDL

CWE Buffer-Overflow Related

- **CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer**
- **CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')**
  - Rank 3 in the Top 25
- **CWE-121: Stack-based Buffer Overflow**
- **CWE-122: Heap-based Buffer Overflow**
- **CWE-124: Buffer Underwrite ('Buffer Underflow')**
- **CWE-125: Out-of-bounds Read**
- **CWE-131: Incorrect Calculation of Buffer Size (!)**
  - Rank 20 in the Top 25
- **CWE-170: Improper Null Termination**
- **CWE-190: Integer Overflow (!)**
  - Rank 24 in the Top 25
- **CWE-193: Off-by-one Error**
- **CWE-805: Buffer Access with Incorrect Length Value**
- **…**

Reference. CWE = Common Weakness Enumeration (http://cwe.mitre.org)

Example: local variable overwrite

- Local variable “authenticate” could be overwritten
- Application control flow could be changed

```
int authenticate(char *username, char *password)
{
  int authenticated;
  char buffer[1024];

  authenticated = verify_password(username, password);

  if (authenticated == 0) {
    sprintf(buffer,
        "password is incorrect for user %s\n", username);
    log("%s", buffer);
  }

  return authenticated;
}
```

Reference. M. Down et al., *The Art of Software Security Assessment*, Addison Wesley, 2012, pg. 176

Example: off-by-one error (1)

Error: wrong array indexing

```
void process_string(char *src)
{
  char dest[32];

  for (i = 0; src[i] && (i <= sizeof(dest)); i++)
    dest[i] = src[i];
}
```

Reference. M. Down et al., *The Art of Software Security Assessment*, Addison Wesley, 2012, pg. 180-181

Example: off-by-one error (2)

Error: wrong string terminator handling

```
int get_user(char *user)
{
  char buf[1024];

  if (strlen(user) >= sizeof(buf))
    die("error: user string too long\n");

  strcpy(buf, user);
}
```

Reference. M. Down et al., *The Art of Software Security Assessment*, Addison Wesley, 2012, pg. 180-181

Example: off-by-one error (3)

Error: wrong string terminator handling

```
int setFilename(char *filename) {
  char name[20];
  sprintf(name, "%16s.dat", filename);
  int success = saveFormattedFilenameToDB(name);
  return success;
}
```

Reference. CWE 193 (http://cwe.mitre.org/data/definitions/193.html)

Example: incorrect length value (1)

Error: wrong size limit considered

```
...
char source[21] = "the character string";
char dest[12];
strncpy(dest, source, sizeof(dest)-1);
dest[sizeof(dest)-1)] = '\0';
...
```

Reference. CWE 805 (http://cwe.mitre.org/data/definitions/805.html)

Example: incorrect length value (2)

- *returnChunkSize()* returns "-1" on error
- the return value is not checked before the *memcpy* operation
- *memcpy()* assumes that the value is unsigned
- when "-1" is returned, it will be interpreted as MAXINT-1 (e.g. 0xFFFFFFFE)

```
int returnChunkSize(void *chunk) {
   /* if chunk info is valid, return the size of usable memory,
    * else, return -1 to indicate an error
    */
 ...
}
int main() {
   ...
   memcpy(destBuf, srcBuf,
          (returnChunkSize(destBuf)-1));
   ...
}
```

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

Reference. CWE 805 (http://cwe.mitre.org/data/definitions/805.html)

Example: incorrect length value (3)

- if *count* is user (attacker) controlled
  - is not checked !!!
- could be given to generate a overflow in the multiplication operation
  - allocates smaller space than accessed

```
bool CopyStructs(InputFile * pInFile,
                 unsigned long count)
{
  unsigned long i;

  m_pStruct = new Structs[count];

  for (i = 0; i < count; i++) {
    if (!ReadFromFile(pInFile, &(m_pStruct[i])))
      break;
  }
}
```

```
new Structs[count] ⇔ malloc(sizeof(Structs) * count);
```

Reference. M. Howard et al., “24 Deadly Sins of Software Security”, 2010, p. 97

Example: incorrect calc. of buffer size (1)

- *malloc(3)* allocates just 3 bytes, instead of space for 3 pointers

```
int *id_sequence;

/* Allocate space for an array of three ids. */

id_sequence = (int*) malloc(3)* sizeof(int*));
if (id_sequence == NULL) exit(1);

/* Populate the id array. */

id_sequence[0] = 13579;
id_sequence[1] = 24680;
id_sequence[2] = 97531;
```

Reference. CWE 131 (http://cwe.mitre.org/data/definitions/131.html)

Example: incorrect calc. of buffer size (2)

- *numHeaders* defined as a signed int
- when assigned a huge unsigned number, it results in a negative number
- when compared, condition is fulfilled
- when used in *malloc*, it is converted back to an unsigned integer => a huge number
- Example
  - numHeaders = -3 (0xFFFFFFFD)
  - numHeaders * sizeof() = -300 (FFFFFED4)
  - malloc(4294966996)

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;

sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders = packet->headers;
if (numHeaders > 100) {
  ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader));
ParsePacketHeaders(packet, headers);
```

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;

sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders = packet->headers;
if (numHeaders > 100 || numHeaders < 0) {
  ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader));
ParsePacketHeaders(packet, headers);
```

Reference. CWE 131 (http://cwe.mitre.org/data/definitions/131.html)

Example: incorrect calc. of buffer size (3)

- when *input* – user controlled
- Problem 1: truncation
  - *strlen()* returns size_t
  - *len* is short
- Problem 2: type casting
  - *len* converted to an (signed) int

```
const long MAX_LEN = 0x7FFF;
char dst[MAX_LEN];

short len = strlen(input);

if (len < MAX_LEN)
     strncpy(dst, input, len);
```

```
size_t strlen(const char *s);
```

Reference. M. Howard et al., “24 Deadly Sins of Software Security”, 2010, p. 121

Example: out-of-bound access (1)

- the buffer index is not validated
- allows access outside the intended area

```
int main (int argc, char **argv) {
  char *items[] = {"boat", "car", "truck", "train"};

  int index = GetUntrustedOffset();

  printf("You selected %s\n", items[index-1]);
}
```

Reference. CWE 125 (http://cwe.mitre.org/data/definitions/125.html)

Example: out-of-bound access (2)

- the buffer index is only checked against the upper limits, but
- not against the lower one (i.e. zero)

```
int getValueFromArray(int *array, int len, int index)
{
  int value;

  if (index < len) {
    value = array[index];
  } else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }

  return value;
}
```

Reference. CWE 125 (http://cwe.mitre.org/data/definitions/125.html)

Example: Improper Null Termination (1)

- *inputbuf* could be not NULL terminated
- *strcpy* could copy more than MAXLEN

```
#define MAXLEN 1024
...
char *pathbuf[MAXLEN];
...
read(cfgfile, inputbuf, MAXLEN); //may not null terminate
strcpy(pathbuf, input_buf); //requires null terminated input
...
```

Reference. CWE 170 (http://cwe.mitre.org/data/definitions/170.html)

Example: Improper Null Termination (2)

- *buf* could be not NULL terminated
- *length* could be greater than MAXPATH

```
char buf[MAXPATH];
char dst[MAXPATH];
...
readlink(path, buf, MAXPATH);
int length = strlen(buf);
...
strncpy(dst, buf, length);
```

Reference. CWE 170 (http://cwe.mitre.org/data/definitions/170.html)

Real-Life Examples

- First well-known Internet worm: *Morris finger worm* (1988)
- Common Vulnerabilities and Exposures (https://cve.mitre.org/find/index.html)
  - Searching string “buffer overflow” → “About 639 results” (actually few thousands)
- Vulnerability Notes Database (https://www.kb.cert.org/vuls/)
  - Searching string “buffer overflow” → “About 240 results”
- Examples
  - CVE-2015-0235 - GHOST: **glibc gethostbyname** buffer overflow
  - CVE-2014-0001 - Buffer overflow in client/mysql.cc in **Oracle MySQL** and MariaDB before 5.5.35
  - CVE-2014-0182 - Heap-based buffer overflow in the virtio_load function in hw/virtio/virtio.c in **QEMU** before 1.7.2
  - CVE-2014-0498 - Stack-based buffer overflow in **Adobe Flash Player** before 11.7.700.269
  - CVE-2014-0513 - Stack-based buffer overflow in **Adobe Illustrator** CS6 before 16.0.5
  - CVE-2014-8271 - **Tianocore UEFI implementation** reclaim function vulnerable to buffer overflow
  - CVE-2013-0002 - Buffer overflow in the **Windows Forms** (aka WinForms) component in Microsoft .NET Framework
  - CVE-2005-3267 - Integer overflow in **Skype** client … leads to a resultant heap-based buffer overflow
  - …

Conclusions

- Buffer Overflow
  - classical, well known, still present (“oldie but goldie”)
  - due to
    - the usage of unsafe function and non-validated user input
    - logic (calculation) errors
- Recommendations for Code Developers
  - Do not use unsafe (string) functions
  - Use the right compiler/linker options
  - Check allocation size calculations
    - Do make size checking
    - Take care of automatic type casting and possible integer overflows
      - use **size_t** (when possible) for allocation size variables
      - take care at casts from **signed to unsigned**
      - ….
- Recommendations for Code Reviewers
  - Check for user input and trace it through the application
  - Check for unsafe functions
  - Check for allocation size calculations