# General Mitigation Against Software Vulnerabilities

# Written Assignment 1

- Due tonight
- Programming assignment 2 will be out soon
- Midterm next Mon. 3/18

# Defensive Coding Practice

# Defensive Coding Practice

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash
  - It's about minimizing trust
- Each module takes responsibility for checking the validity of all inputs sent to it
  - Even if you "know" your callers will never send a NULL pointer
  - Better to throw an exception than run malicious code

# How to Program Defensively

- Code reviews, real or imagined
  - Organize your code so it is obviously correct
  - Re-write until it would be self-evident to a reviewer

- Remove the opportunity for programmer mistakes with better languages and libraries
  - Java performs automatic bounds checking
  - C++ provides a safe `std::string` class

# Secure Coding Practices

- Think about all potential inputs, no matter how peculiar

```
char digit_to_char(int i) {
char convert[] = "0123456789";
return convert[i];
}
```

# Secure Coding Practices

- Think about all potential inputs, no matter how peculiar

```
char digit_to_char(int i) {
char convert[] = "0123456789";
return convert[i];
}
```

```
char digit_to_char(int i) {
   char convert[] = "0123456789";
   if(i < 0 || i > 9)
      return '?';
   return convert[i];
}
```

# Secure Coding Practices

- Use safe string functions
  - Traditional string library routines assume target buffers have sufficient length

```
char str[4];
char buf[10] = "good";
strcpy(str,"hello"); // overflows str
strcat(buf," day to you"); // overflows buf
```

- Safe versions check the destination length

```
char str[4];
char buf[10] = "good";
strlcpy(str,"hello",sizeof(str)); //safer null terminated
strlcat(buf," day to you",sizeof(buf));
```

# Secure Coding Practices

- for string-oriented functions
  - `strcat` ⟹ `strlcat`
  - `strcpy` ⟹ `strlcpy`
  - `strncat` ⟹ `strlcat`
  - `strncpy` ⟹ `strlcpy`
  - `sprintf` ⟹ `snprintf`
  - `vsprintf` ⟹ `vsnprintf`
  - `gets` ⟹ `fgets`
  - `strn`… do not NUL-terminate if they run up against the size limit
  - `strl`… are not standard library functions

# Secure Coding Practices

- Use safe string library
  - Safety first, despite some performance loss
- Example: Very Secure FTP(vsftp) string library

```
struct mystr; // impl hidden
void str_alloc_text(struct mystr* p_str, const char* p_src);
void str_append_str(struct mystr* p_str, const struct mystr* p_other);
int str_equal(const struct mystr* p_str1, const struct mystr* p_str2);
int str_contains_space(const struct mystr* p_str);
…
```

# Secure Coding Practices

- Understand pointer arithmetic

```
int *search(int *p, int val) {
   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

# Secure Coding Practices

- Understand pointer arithmetic

```
int *search(int *p, int val) {
   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

- Use the right unit

```
int *search(int *p, int val) {
   while (p && *p != val)
      p += 1;

   return p;
}
```

# Secure Coding Practices

- Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int *)); //may reuse p's space
*q = &x;
*p = 5;  // q may point to address location 5
**q = 3; //crash (or worse)! segmentation fault
```

# Secure Coding Practices

- Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int *)); //may reuse p's space
*q = &x;
*p = 5; //crash, but in a good way
**q = 3;
```

# Secure Coding Practice

- Manage memory properly
  - Common approach in C: goto error handling code
  - Like try/finally in languages like Java

```
int foo(int arg1, int arg2) {
   struct foo *pf1, *pf2;
   int retc = -1; pf1 = malloc(sizeof(struct foo));
   if (!isok(arg1)) goto CLEANUP;
   ...
   pf2 = malloc(sizeof(struct foo)); if (!isok(arg2)) goto FAIL_ARG2;
   ...
   FAIL_ARG2:
   free(pf2); //fallthru
   CLEANUP: free(pf1); return retc;
}
```

# Secure Coding Practice

- Use a safe allocator
  - ASLR challenges exploits by making the base address of libraries unpredictable
  - Challenge heap-based overflows by making the addresses returned by malloc unpredictable
    - Can have some negative performance impact
  - Example:
    - DieHard: approximating an infinite heap

# Secure Coding Practice

- Favor safe libraries
  - Libraries encapsulate well-thought-out design
    - Take advantage
  - Smart pointers
    - Pointers with only safe operations
    - Lifetimes managed appropriately
    - First in Boost library, now a C++ standard
  - Networking: Google protocol buffers, Apache Thrift
    - For dealing with network-transmitted data
    - Ensures input validation, parsing
    - Efficient

# Automated Testing Techniques

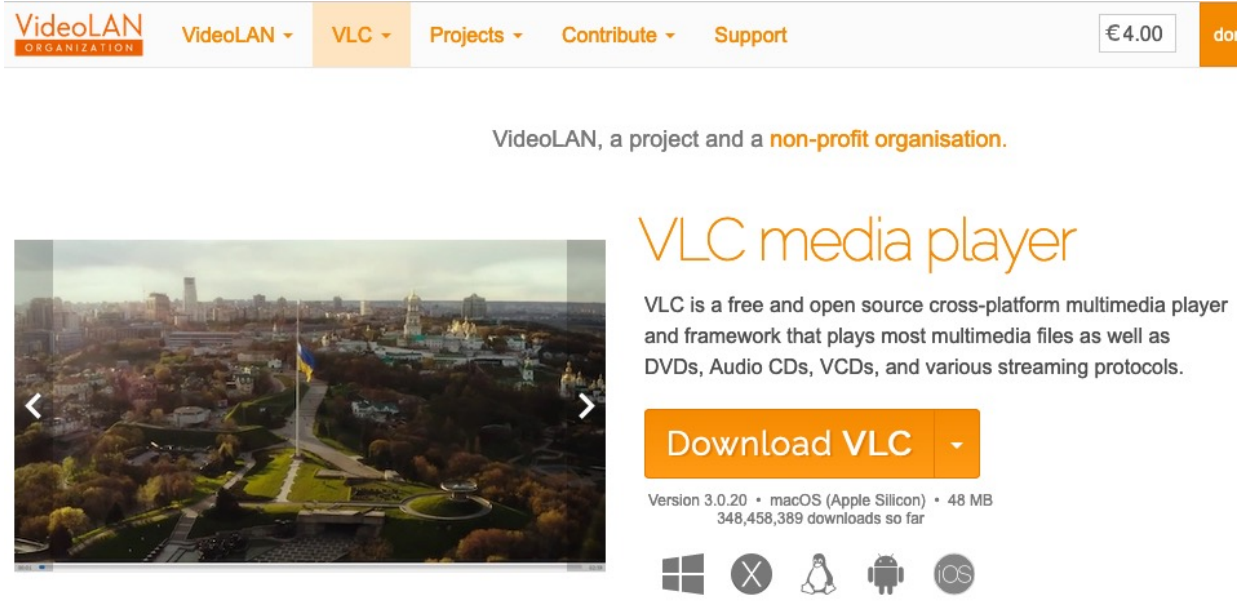
# Automated Testing Techniques

- Code analysis
  - Static: Many of the bugs we've shown could be easily detected
  - Dynamic: Run in a VM and look for bad writes(Valgrind)

- Fuzz testing
  - Generate many random inputs, see if the program fails
    - Totally random
    - Start with a valid input file and mutate
    - Structure-driven input generation: take into account the intended format of the input
  - Typically involves many inputs

# Static analysis - Walk you through an example

- Try to hack VideoLan's popular VLC media player
  - VLC 0.9.4 on Windows Vista SP1(32-bit)



# Phase I: Vulnerability Discovery

- Step 1: Generate a list of the demuxers of VLC
- Step 2: Identify the input data
- Step 3: Trace the input data
- In digital video, **demuxing** or **demultiplexing** *refers to the process of separating audio and video as well as other data from a video stream or container in order to play the file. A demuxer is software that extracts the components of such a stream or container.*

# Step 1: Generate a List of Demuxers of VLC


Directory: vlc-0.9.4\modules\demux\

# Step 2: Identify the input data

**Source code file** *vlc-0.9.4\include\vlc_demux.h*

```
[..]
41 struct demux_t
42 {
43     VLC_COMMON_MEMBERS
44
45     /* Module properties */
46     module_t    *p_module;
47
48     /* eg informative but needed (we can have access+demux) */
49     char        *psz_access;
50     char        *psz_demux;
51     char        *psz_path;
52
53     /* input stream */
54     stream_t    *s;     /* NULL in case of a access+demux in one */
[..]
```

A reference to the input data to be demuxed

# Step 3: Trace the input data

**Source code file** *vlc-0.9.4\modules\demux\Ty.c*

**Function** parse_master()

```
[..]
1623 static void parse_master(demux_t *p_demux)
1624 {
1625     demux_sys_t *p_sys = p_demux->p_sys;
1626     uint8_t mst_buf[32];
1627     int i, i_map_size;
1628     int64_t i_save_pos = stream_Tell(p_demux->s);
1629     int64_t i_pts_secs;
1630
1631     /* Note that the entries in the SEQ table in the stream may have
1632        different sizes depending on the bits per entry.  We store them
1633        all in the same size structure, so we have to parse them out one
1634        by one.  If we had a dynamic structure, we could simply read the
1635        entire table directly from the stream into memory in place. */
```

# Trace the input data(cont.)

```
1636
1637      /* clear the SEQ table */
1638      free(p_sys->seq_table);
1639
1640      /* parse header info */
1641      stream_Read(p_demux->s, mst_buf, 32);
1642      i_map_size = U32_AT(&mst_buf[20]);  /* size of bitmask, in bytes */
1643      p_sys->i_bits_per_seq_entry = i_map_size * 8;
1644      i = U32_AT(&mst_buf[28]);   /* size of SEQ table, in bytes */
1645      p_sys->i_seq_table_size = i / (8 + i_map_size);
1646
1647      /* parse all the entries */
1648      p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
1649      for (i=0; i<p_sys->i_seq_table_size; i++) {
1650          stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
[..]
```

# Trace the input data(cont.)

```
1636
1637    /* clear the SEQ table */
1638    free(p_sys->seq_table);
1639
1640    /* parse header info */
1641    stream_Read(p_demux->s, mst_buf, 32);
1642    i_map_size = U32_AT(&mst_buf[20]);  /* size of bitmask, in bytes */
1643    p_sys->i_bits_per_seq_entry = i_map_size * 8;
1644    i = U32_AT(&mst_buf[28]);   /* size of SEQ table, in bytes */
1645    p_sys->i_seq_table_size = i / (8 + i_map_size);
1646
1647    /* parse all the entries */
1648    p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
1649    for (i=0; i<p_sys->i_seq_table_size; i++) {
1650        stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
[..]
```

Copied 32 bytes from user input file to the stack buffer `mst_buf`

# Trace the input data(cont.)

```
1636
1637    /* clear the SEQ table */
1638    free(p_sys->seq_table);
1639
1640    /* parse header info */
1641    stream_Read(p_demux->s, mst_buf, 32);
1642    i_map_size = U32_AT(&mst_buf[20]);  /* size of
1643    p_sys->i_bits_per_seq_entry = i_map_size * 8;
1644    i = U32_AT(&mst_buf[28]);    /* size of SEQ tab
1645    p_sys->i_seq_table_size = i / (8 + i_map_size);
1646
1647    /* parse all the entries */
1648    p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
1649    for (i=0; i<p_sys->i_seq_table_size; i++) {
1650        stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
[..]
```

Copied 32 bytes from user input file to the stack buffer `mst_buf`

4 bytes user-controlled data Extracted from the buffer and stored in `i_map_size`

# Trace the input data(cont.)

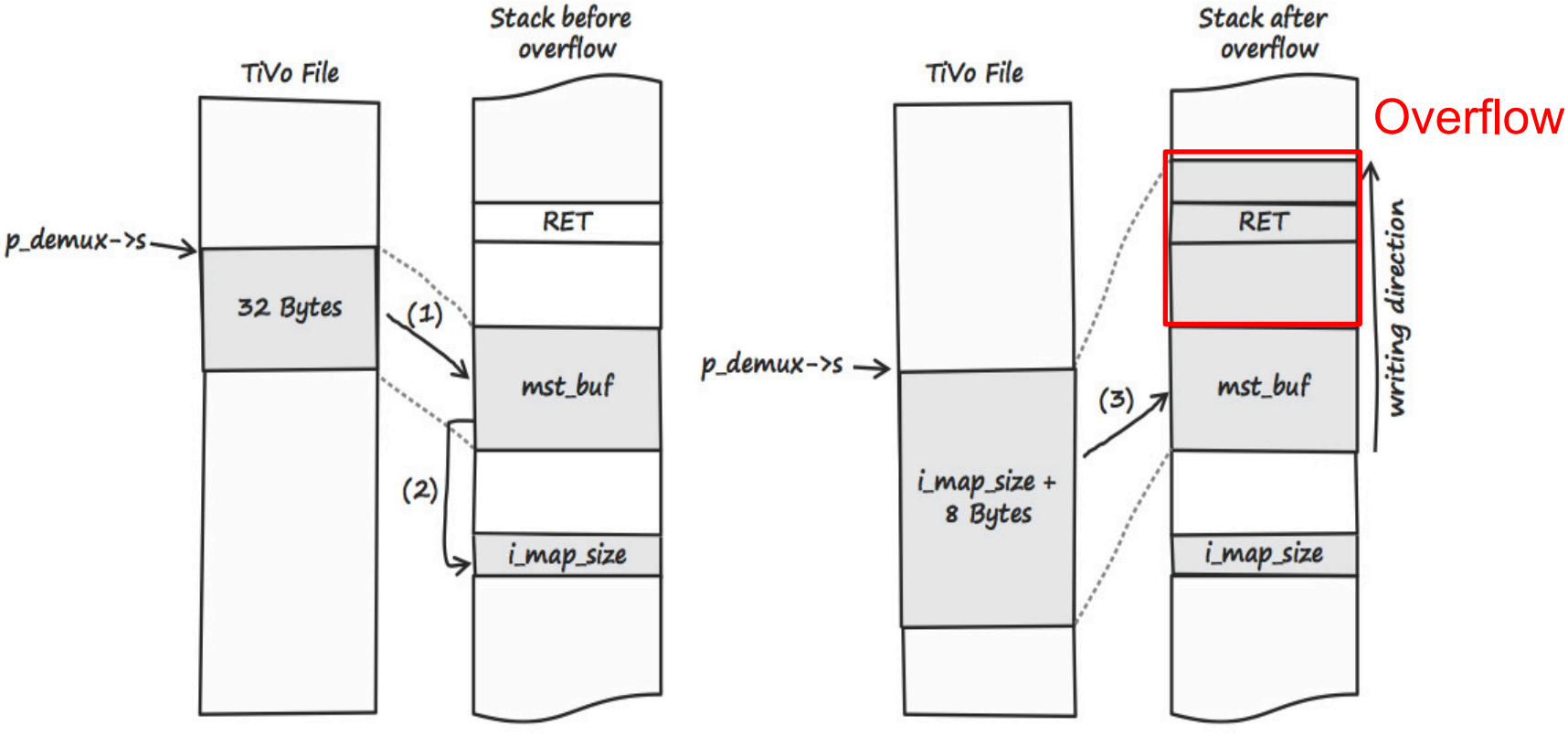
```
1636
1637    /* clear the SEQ table */
1638    free(p_sys->seq_table);
1639
1640    /* parse header info */
1641    stream_Read(p_demux->s, mst_buf, 32);
1642    i_map_size = U32_AT(&mst_buf[20]);  /* size of
1643    p_sys->i_bits_per_seq_entry = i_map_size * 8;
1644    i = U32_AT(&mst_buf[28]);    /* size of SEQ tab
1645    p_sys->i_seq_table_size = i / (8 + i_map_size);
1646
1647    /* parse all the entries */
1648    p_sys->seq_table = malloc(p_sys->i_seq_table_size * si
1649    for (i=0; i<p_sys->i_seq_table_size; i++) {
1650        stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
[..]
```

Copied 32 bytes from user input file to the stack buffer `mst_buf`

4 bytes user-controlled data Extracted from the buffer and stored in `i_map_size`

User-controlled data is copied once again in stack buffer `mst_buf`. If `i_map_size > 24`, buffer overflow will occur.

# Illustration of buffer overflow



# Phase II: Exploitation

- Step 1: Find a sample TiVo movie file
- Step 2: Find a code path to reach the vulnerable code
- Step 3: Manipulate the TiVo movie file to crash VLC
- Step 4: Manipulate the TiVo movie file to gain control of EIP

# Step 1: Find a sample TiVo movie file

```
$ wget http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
--2008-10-12 21:12:25--  http://samples.mplayerhq.hu/TiVo/test-dtivo-junkskip.ty%2b
Resolving samples.mplayerhq.hu... 213.144.138.186
Connecting to samples.mplayerhq.hu|213.144.138.186|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5242880 (5.0M) [text/plain]
Saving to: `test-dtivo-junkskip.ty+´

100%[=========================>] 5,242,880    240K/s   in 22s

2008-10-12 21:12:48 (232 KB/s) - `test-dtivo-junkskip.ty+´ saved [5242880/5242880]
```

Website http://samples.mplayerhq.hu is a good starting point to search for all kinds of multimedia file-format samples

# Step 2: Find a code path to reach the vulnerable code

```
1866    /* check if it's a PART Header */
1867    if( U32_AT( &p_peek[ 0 ] ) == TIVO_PES_FILEID )
1868    {
1869        /* parse master chunk */
1870        parse_master(p_demux);
1871        return get_chunk_header(p_demux);
1872    }
```

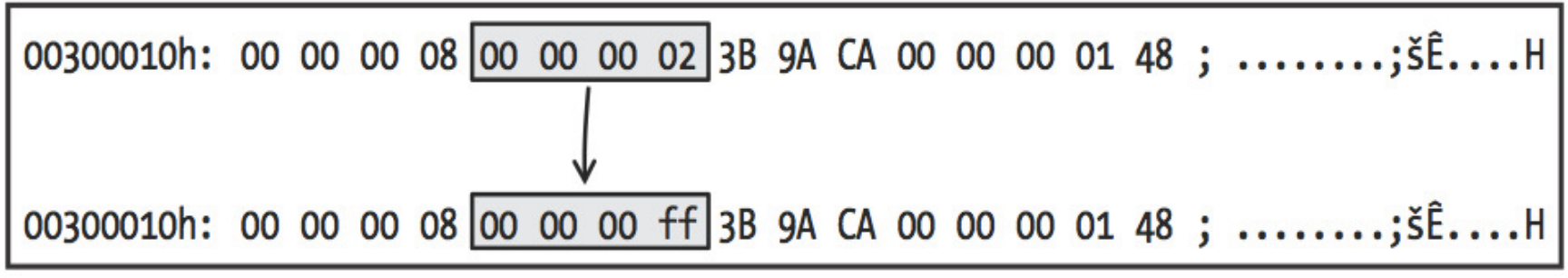
```
[..]
 112 #define TIVO_PES_FILEID    ( 0xf5467abd )
[..]
```

Pattern in the sample file

```
00300000h: F5 46 7A BD 00 00 00 02 00 02 00 00 00 01 F7 04 ; õFz½..........÷.
00300010h: 00 00 00 08 00 00 00 02 3B 9A CA 00 00 00 01 48 ; ........;šÊ....H
```

# Step 3: Manipulate the TiVo movie file to crash VLC

- Change the 4-byte value at the sample file offset of `i_map_size` (which is 0x00300014 in the file)

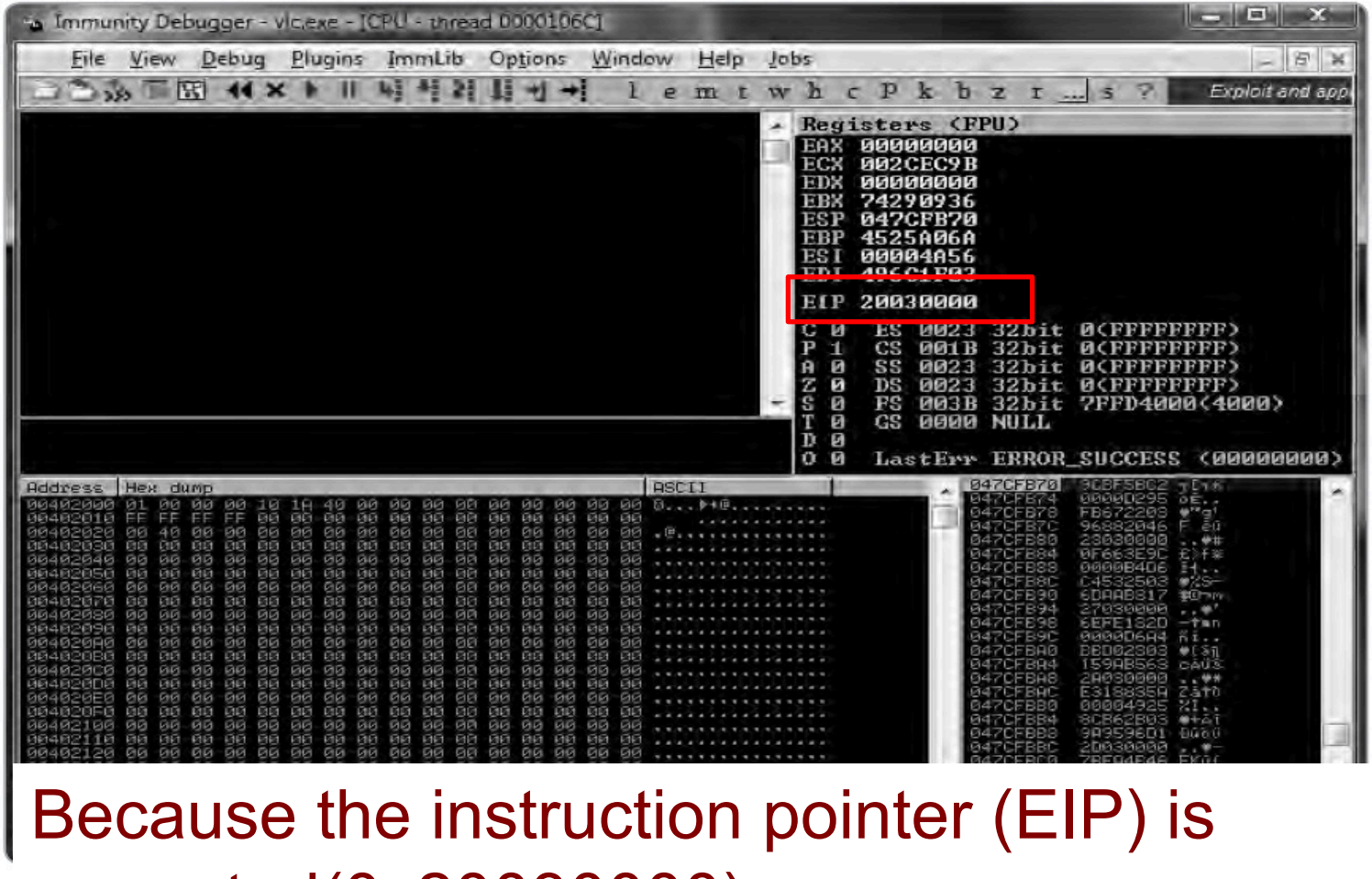



Change `i_map_size` to 0xff(255)

Enough to overflow the 32-byte stack buffer

VLC Crashed!!!

# VLC access violation in immunity debugger

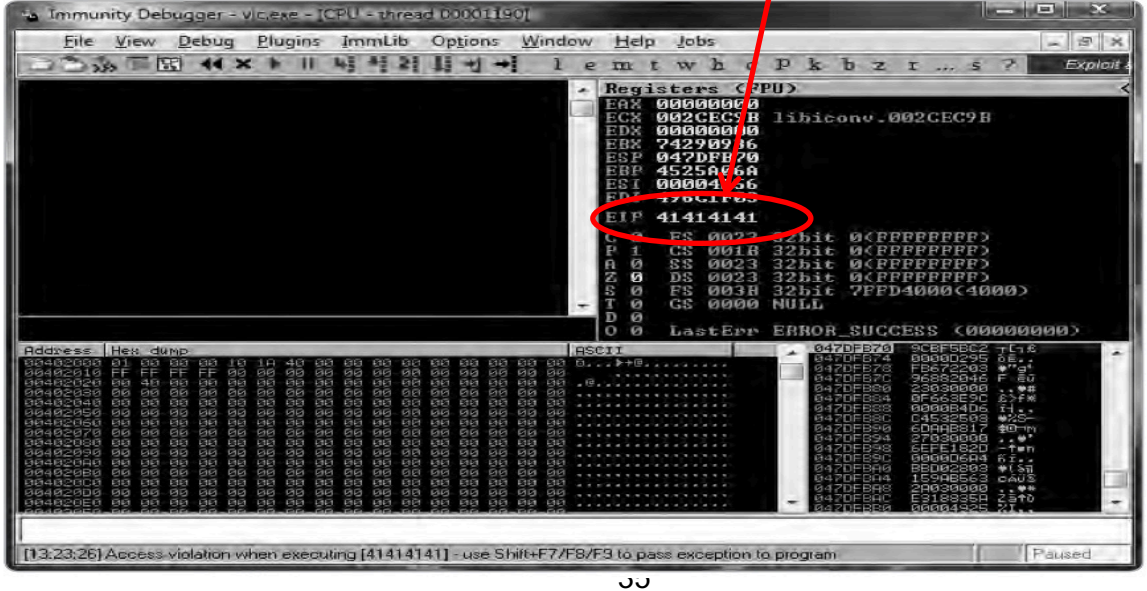



Because the instruction pointer (EIP) is corrupted(0x20030000)

# Step 4: Manipulate the TiVo movie file to gain control of EIP

```
00300050h: 56 4A 00 00 03 1F 6C 49 6A A0 25 45 [00 00 03 20] ; VJ....lIj %E...
                                               ↓
00300050h: 56 4A 00 00 03 1F 6C 49 6A A0 25 45 [41 41 41 41] ; VJ....lIj %EAAAA
```

Exploit found



# Phase III: Vulnerability remediation

- Choices after found a vulnerability
  - Coordinated disclosure
    - Sell it to vulnerability broker
  - Full disclosure
    - Release to public without notifying the vendor
  - Notified the VLC maintainers, provided them with the necessary information, and coordinated with them on the timing of public disclosure.

# First patch from VLC

```
--- a/modules/demux/ty.c
+++ b/modules/demux/ty.c
@@ -1639,12 +1639,14 @@ static void parse_master(demux_t *p_demux)
     /* parse all the entries */
     p_sys->seq_table = malloc(p_sys->i_seq_table_size * sizeof(ty_seq_table_t));
     for (i=0; i<p_sys->i_seq_table_size; i++) {
-        stream_Read(p_demux->s, mst_buf, 8 + i_map_size);
+        stream_Read(p_demux->s, mst_buf, 8);
         p_sys->seq_table[i].l_timestamp = U64_AT(&mst_buf[0]);
         if (i_map_size > 8) {
             msg_Err(p_demux, "Unsupported SEQ bitmap size in master chunk");
+            stream_Read(p_demux->s, NULL, i_map_size);
             memset(p_sys->seq_table[i].chunk_bitmask, i_map_size, 0);
         } else {
+            stream_Read(p_demux->s, mst_buf + 8, i_map_size);
             memcpy(p_sys->seq_table[i].chunk_bitmask, &mst_buf[8], i_map_size);
         }
```

- The formerly vulnerable call to stream_Read() now uses a fixed size value(8), and the user-controlled value of `i_map_size` is used only as a size value for stream_Read() if it is less than or equal to 8.

# But…

- The variable `i_map_size` is still of the type signed int

**Source code file** *vlc-0.9.4\modules\demux\Ty.c*
**Function** parse_master()

```
[..]
1623 static void parse_master(demux_t *p_demux)
1624 {
1625     demux_sys_t *p_sys = p_demux->p_sys;
1626     uint8_t mst_buf[32];
1627     int i, i_map_size;
1628     int64_t i_save_pos = stream_Tell(p_demux->s);
1629     int64_t i_pts_secs;
1630
1631     /* Note that the entries in the SEQ table in the stream may have
1632        different sizes depending on the bits per entry.  We store them
1633        all in the same size structure, so we have to parse them out one
1634        by one.  If we had a dynamic structure, we could simply read the
1635        entire table directly from the stream into memory in place. */
```

- If a value >= 0x80000000 is supplied for i_map_size, it's interpreted as negative(type overflow), and the overflow will still occur in the stream_Read() and memcpy() functions of the else branch of the patch.

# Final patch

```
[..]
@@ -1616,7 +1618,7 @@ static void parse_master(demux_t *p_demux)

 {
     demux_sys_t *p_sys = p_demux->p_sys;
     uint8_t mst_buf[32];
-    int i, i_map_size;
+    uint32_t i, i_map_size;
     int64_t i_save_pos = stream_Tell(p_demux->s);
     int64_t i_pts_secs;
[..]
```

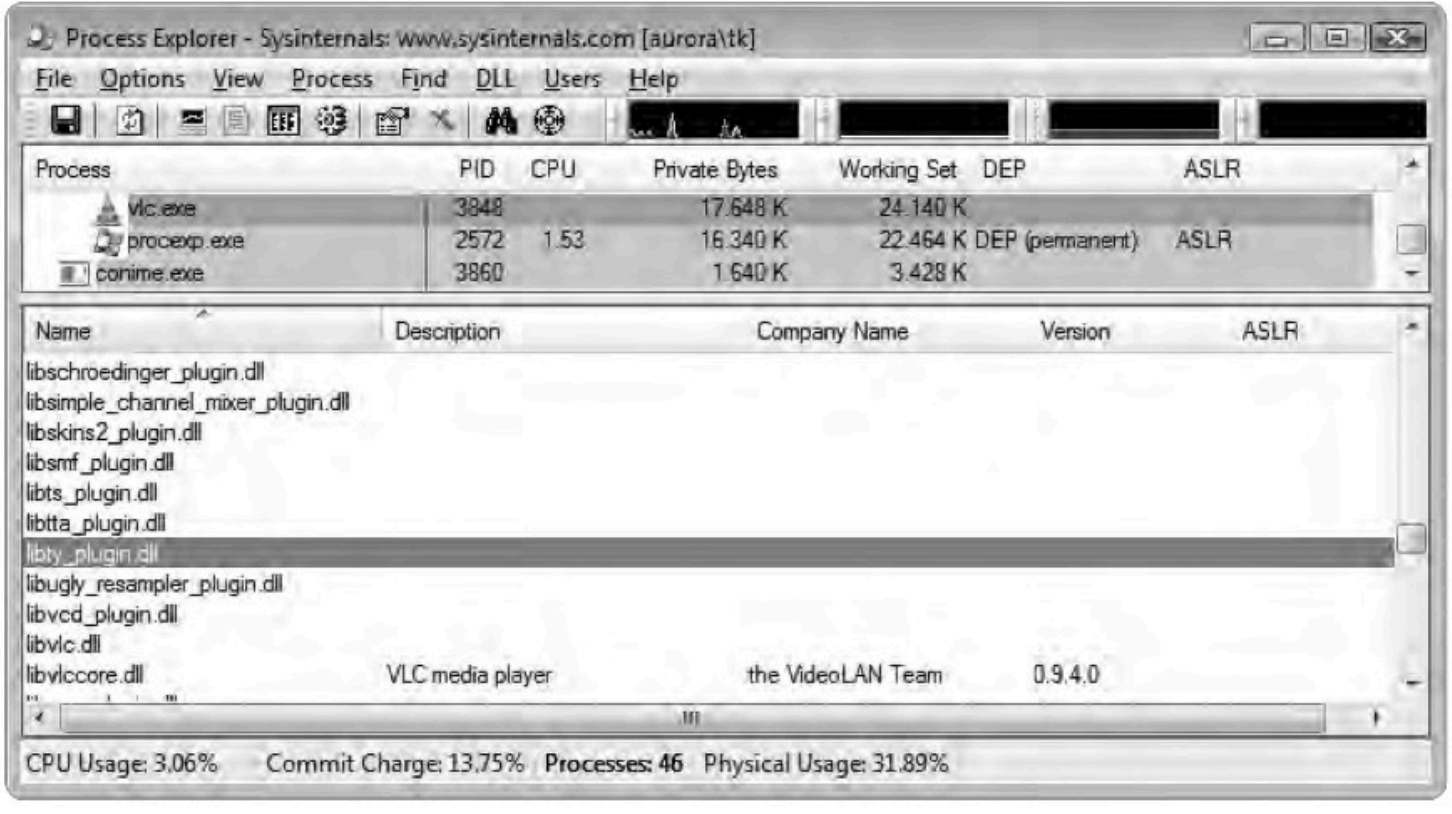
Signed int changed to unsigned int

# Microsoft Windows has security policies…

- Microsoft Windows Vista has security cookie or /GS feature (i.e., canary)

- And ASLR or NX/DEP (NX: Non-Executable, DEP: Data Execution Prevention) in Microsoft Windows Vista could have prevented arbitrary code execution on stack

# DEP policy is OptIn in the default Vista




The process didn't OptIn for DEP

# VLC compilation

- VLC compiled using Cygwin
  - set of utilities designed to provide the look and feel of Linux within the Windows operating system.
- Since the linker switches(DEP/ASLR), are supported only by Microsoft's Visual C++ 2005 SP1 and later (and thus are not supported by Cygwin), they aren't supported by VLC.

# Vulnerability released

- The bug was assigned CVE-2008-4654.
- CVE: Common Vulnerabilities and Exposures
  - Provides a reference-method for publicly known information-security vulnerabilities and exploits
  - Maintained by MITRE Corporation

| CVE-ID | |
|---|---|
| **CVE-2008-4654** | Learn more at National Vulnerability Database (NVD)<br>• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings |
| **Description** | |
| Stack-based buffer overflow in the parse_master function in the Ty demux plugin (modules/demux/ty.c) in VLC Media Player 0.9.0 through 0.9.4 allows remote attackers to execute arbitrary code via a TiVo TY media file with a header containing a crafted size value. | |

# Lessons learned

- Never trust user input (this includes file data, network data, etc.)
  - **Attack surface** of the program
- Never use unvalidated length or size values
- Always make use of the exploit mitigation techniques offered by modern operating systems wherever possible.

# Fuzzing

- A software testing technique that involves providing invalid, unexpected, or random data to the inputs of a computer program
- A form of penetration testing
  - **Actively** trying to find exploitable vulnerabilities
  - Useful for both attacker and defenders
- Automated or semi-automated
- Mostly for hunting memory corruption style
- Many bugs come from this way these days (could be over 50%)

# Black-box Fuzz Testing

- Given a program, knows nothing about it or its input, simply feed it random inputs, see whether it crashes
- Advantage: easy to use
- Disadvantage: inefficient
  - Input often requires structures, random inputs are likely to be malformed
  - Inputs that would trigger a crash is a very small fraction, probability of getting lucky may be very low

# Example of black-box fuzzing

- Random mutation and run cases
- You can do this within only 10 line python

```
import os, sys, random
def go():
  return random.randrange(0, 0x100)
filesize = os.path.getsize("./sample.xxx")
fp = open("./sample.xxx", "rb++")
tmpoffset = random.randrange(0, filesize)
fp.seek(tmpoffset, 0)
fp.write("%c%c%c%c" % (go(), go(), go(), go()))
fp.close()
os.system("target_binary sample.xxx")
```

# Enhancement I: Mutation-Based Fuzzing

- Take a **well-formed input**, randomly perturb (flipping bit, etc.)
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random
- Examples:
  - E.g., ZZUF, very successful at finding bugs in many real-world programs, http://sam.zoy.org/zzuf/

# Example: fuzzing a pdf viewer

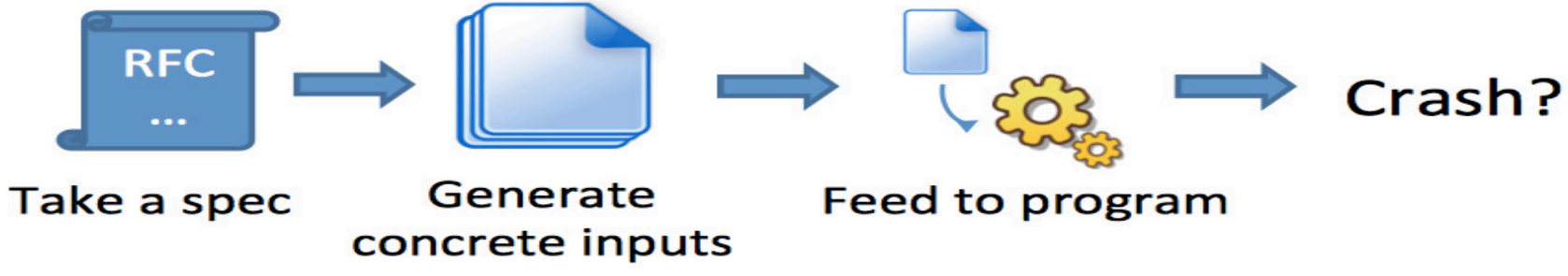
- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus of PDF files
- Use fuzzing tool (or script)
  - Grab a file
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

# Mutation-based Fuzzing In Short

| | | | | |
|---|---|---|---|---|
| Mutation-based | Super easy to setup and automate  | Little to no protocol knowledge required  | Limited by initial corpus  | May fail for protocols with checksums, those which depend on challenge  |

# Enhancement II: Generation-Based Fuzzing

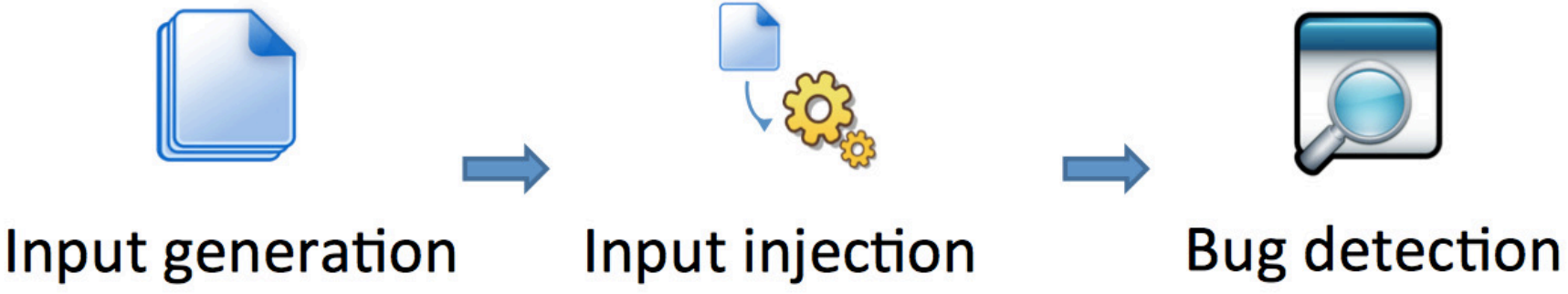
- Test cases are generated from some description of the format: RFC, documentation, a grammar, etc.
  - Using specified protocols/file format info
  - E.g., the SPIKE fuzzer
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



# Generation-Based Fuzzing In Short

| | | | | |
|---|---|---|---|---|
| Mutation-based | Super easy to setup and automate + | Little to no protocol knowledge required + | Limited by initial corpus − | May fail for protocols with checksums, those which depend on challenge − |
| Generation-based | Writing generator can be labor intensive for complex protocols − | Have to have spec of protocol (Often can find good tools for existing protocols e.g. http, SNMP) − | Completeness + | Can deal with complex dependencies e.g. checksums + |

# Fuzzing Tools & Frameworks



# Input Generation

- Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)
  - MuDynamics, FTPFuzz, WebScarab
- Fuzzing Frameworks: providing a fuzz set with a given spec
  - SPIKE, Peach, Sulley
- Mutation-based fuzzers
  - Taof, GPF, ProxyFuzz, PeachShark
- Special purpose fuzzers
  - Regular expressions, etc.

# Input injection

- Simplest
  - Replay fuzzed packet trace
- Modify existing program/client
  - Invoke fuzzer at appropriate point
- Use fuzzing framework
  - e.g. Peach automates generating fuzzers

# Bug Detection

- See if program crashed
  - Type of crash can tell a lot(SEGV vs. assert fail)
- Run program under dynamic memory error detector (Valgrind/purify)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Write your own checker: e.g. Valgrind tools

# How Much Fuzzing Is Enough?

- Mutation based fuzzers may generate an infinite number of test cases...
  - When has the fuzzer run long enough?

- Generation based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?

# Code Coverage

- Some of the answers to these questions lie in *code coverage*
- **Code coverage** is a metric which can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov

# Line Coverage

- Line/block coverage: Measures how many lines of source code have been executed.
- For the following code, only add() tested but not subtract().

```
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int main() {
   int sum = add(5, 3);
   return 0;
}
```

# Branch Coverage

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the following code, `if` branch is covered but not `else` branch. Use negative inputs to reach 100% coverage.

```
int checkPositive(int number) {
    if (number > 0) {
        return 1; // Positive
    } else {
        return 0; // Non-positive
    }
}
int main() {
   int check = checkPositive(5);
   return 0;
}
```

# Path Coverage

- Path coverage: Measures how many paths have been taken.
- For the following code, 100% path coverage achieved

```
int calculate(int a, int b, bool addFlag) {
    if (addFlag) {
        return a + b;
    } else {
        return a - b;
    }
}
int main() {
   int result1 = calculate(5, 3, true);
   int result2 = calculate(5, 3, false);
   return 0;
}
```

# Branch vs Path Coverage

- For the following example
  - Branch coverage only needs 2 pair of test case, e.g. a=true, b=true and a=false, b=false
  - Path coverage needs 4 pairs of test case,
    - a=true, b=true
    - a=true, b=false
    - a=false, b=true
    - a=false, b=false

```
void exampleFunction(bool a, bool b) {
    if (a) {
        // Branch 1
    } else {
        // Branch 2
    }
    if (b) {
        // Branch 3
    } else {
        // Branch 4
    }
}
```

# Coverage-guided Fuzzing

- Code Coverage: Aim to increase code coverage with every new input
- Input Generation: Starting from a feasible input, the fuzzer modifies it based on algorithm to explore new paths
- Feedback loop: fuzzer analyzes the coverage achieved by input and use this information to craft new inputs
- Bug detection: can uncover various types of bugs
- Efficient than blind fuzzing
- Tools: American Fuzzy Lop(AFL) , Libfuzzer from LLVM

# Problems of code coverage

- For:

```
mySafeCpy(char *dst, char* src){
 if(dst && src)
    strcpy(dst, src);
}
```

- Does full line coverage guarantee finding the bug?
- Does full branch coverage guarantee finding the bug?

- Libfuzzer integrate coverage-guided fuzzing with memory sanitizor like AddressSanitizer(Asan) to find runtime memory bugs

# Fuzzing Rules of Thumb

- Protocol specific knowledge very helpful
- Generational tends to beat random, better spec's make better fuzzers
- More fuzzers is better
  - Each implementation will vary, different fuzzers find different bugs
- The longer you run, the more bugs you may find
- Best results come from guiding the process
- Code coverage can be very useful for guiding the process: American Fuzzy Lop(AFL)

# Recap: Automated Testing Techniques

- Code analysis
  - Static: Many of the bugs we've shown could be easily detected
  - Dynamic: Run in a VM and look for bad writes(Valgrind)

- Fuzz testing
  - Generate many random inputs, see if the program fails
    - Totally random
    - Start with a valid input file and mutate
    - Structure-driven input generation: take into account the intended format of the input
  - Typically involves many inputs

# Symbolic Execution

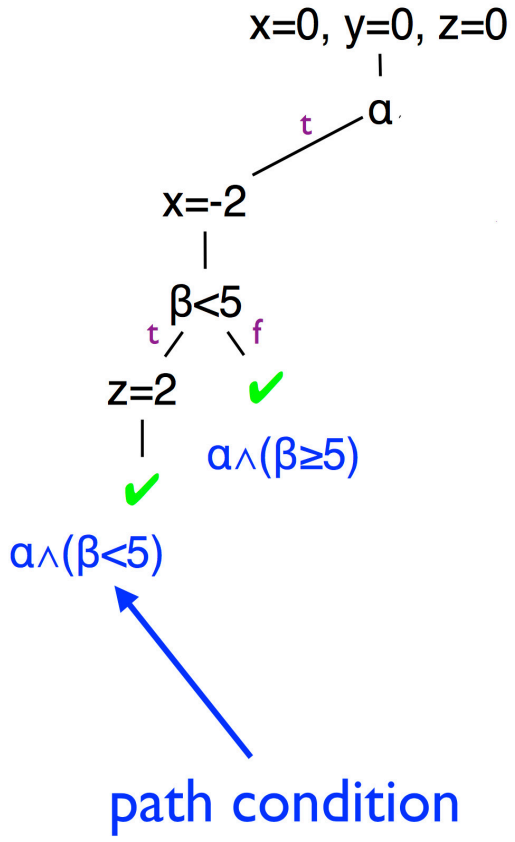
- King, CACM 1976.
  - Key idea: generalize testing by using unknown or symbols
- Symbolic execution: a program analysis technique that executes a program with **symbolic inputs** instead of concrete input values.
  - Aim for **comprehensive** examination of program **paths**
- Symbolic variables in evaluation
  - Symbolic executor executes program, tracking *symbolic state*.
- If execution path depends on unknown/symbols, we fork symbolic executor
  - at least, conceptually

# Symbolic execution example

```
1. int a = α, b = β, c = γ;
2.                  // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```

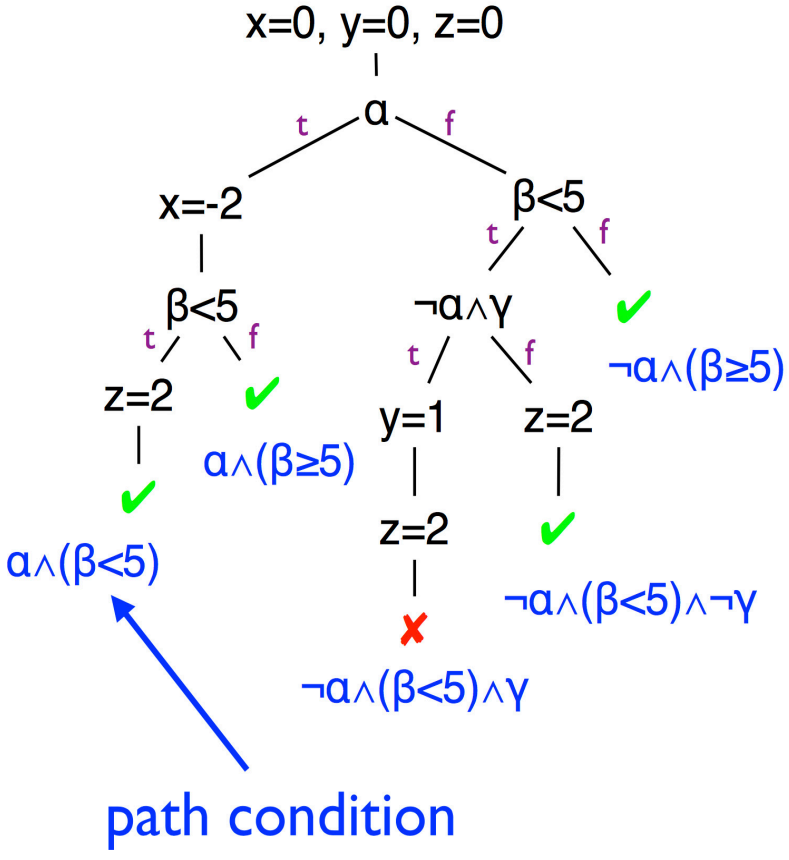
# Symbolic execution example

```
1. int a = α, b = β, c = γ;
2.                  // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```



# Symbolic execution example

```
1. int a = α, b = β, c = γ;
2.                  // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```

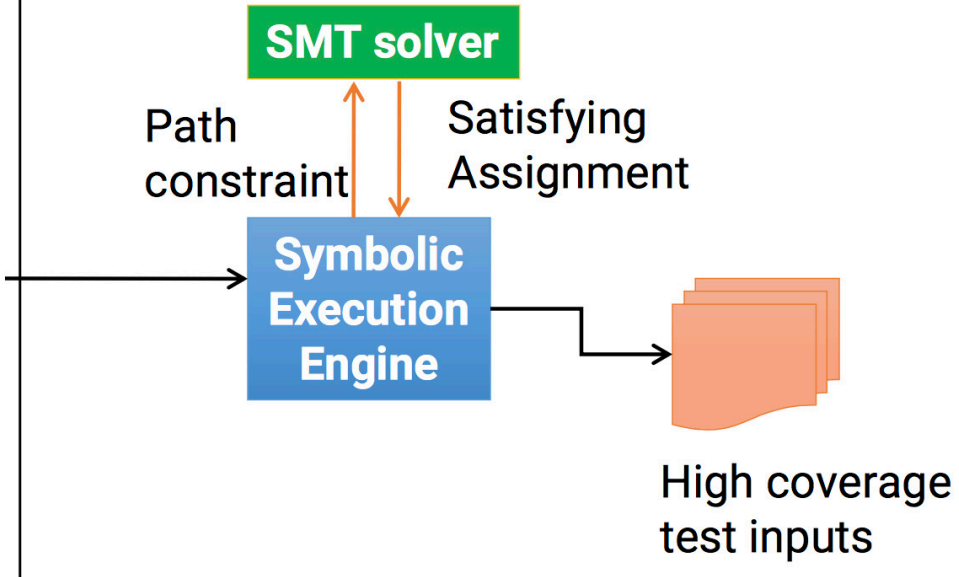


# What's going on here?

- During symbolic execution, we are trying to determine if certain formulas are **satisfiable**
  - E.g., is a particular program point reachable?
    - Figure out if the path condition is satisfiable
  - E.g., is array access a[i] out of bounds?
    - Figure out if conjunction of path condition and i<0 OR i > a.length is satisfiable
  - E.g., generate concrete inputs that execute the same paths
- This is enabled by powerful SMT/SAT solvers
  - SAT = Satisfiability
  - SMT = Satisfiability modulo theory = SAT++

# Symbolic execution for software testing

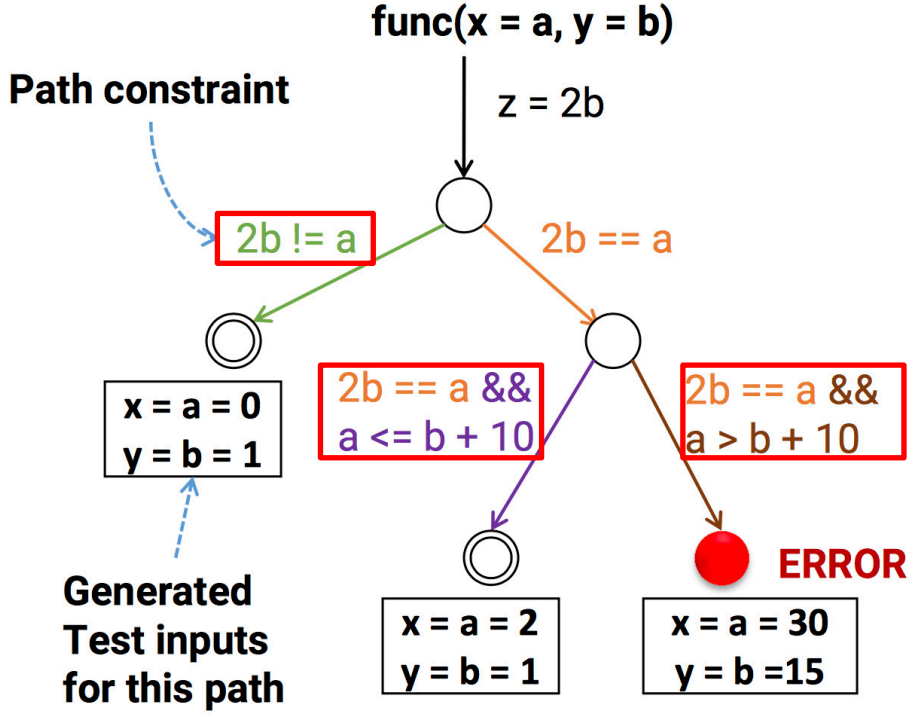
```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
            if (x > y + 10)
                ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
```




**Symbolic Execution**

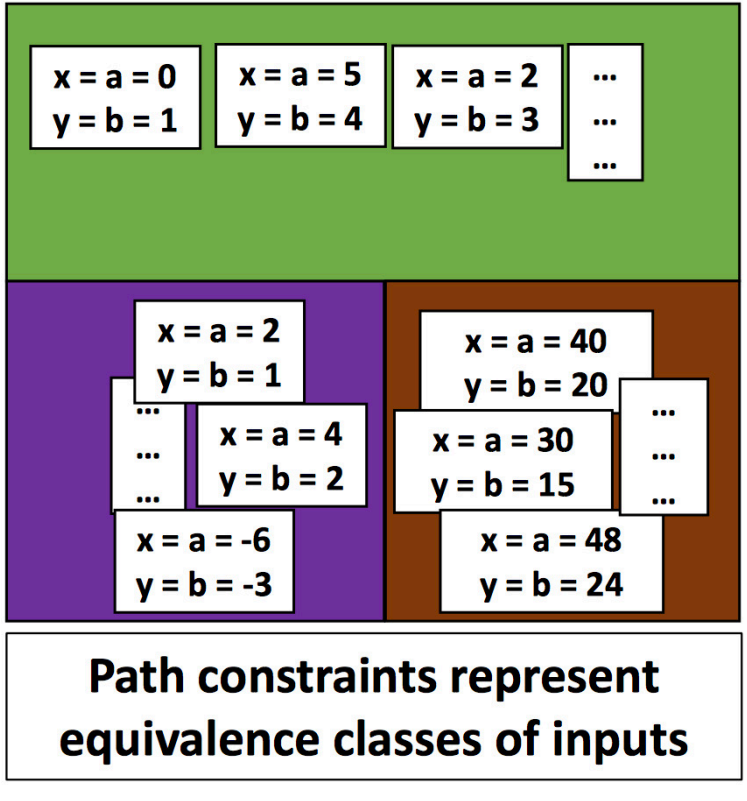
# How does symbolic execution work?

```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
            if (x > y + 10)
                ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
```

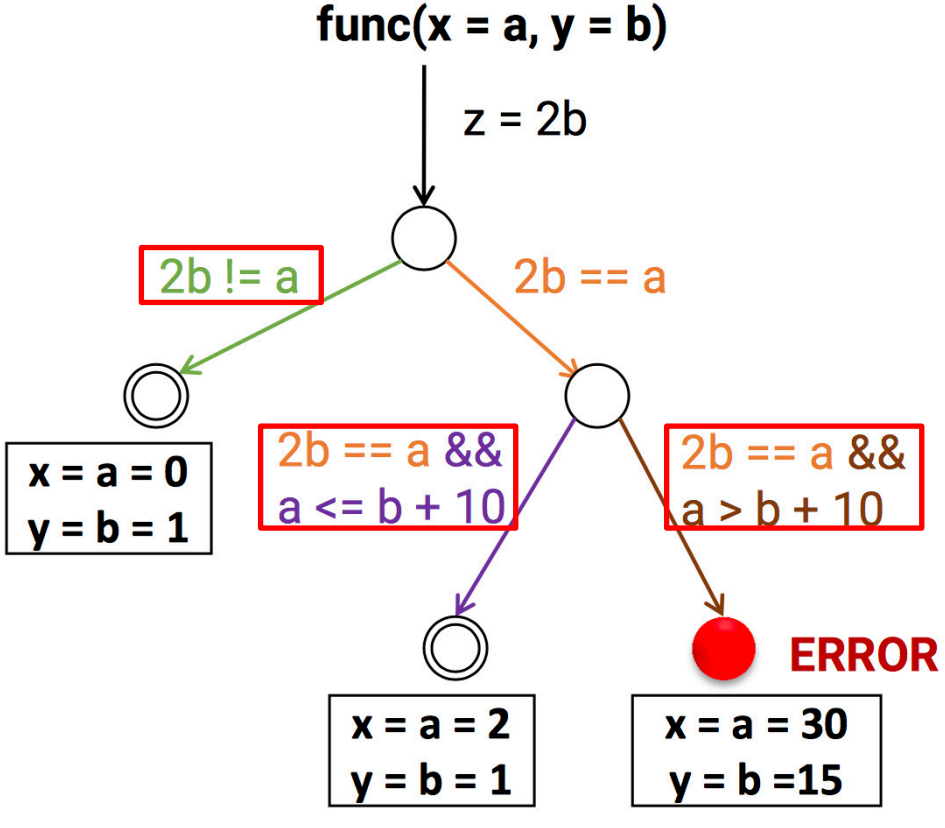



**Note: Require inputs to be marked as symbolic**

# Equivalence classes of inputs




## How does symbolic execution work?



# Summary of symbolic execution

- Execute the program with symbolic valued inputs (Goal: **theoretically 100% path coverage**)
- Represents equivalence class of inputs with path constraints (first order logic formulas)
- One path constraint abstractly represent all inputs that induces the program execution to go down a specific path
- Solve the path constraint to obtain one representative input that exercises the program to go down that specific path