Introduction of Software Security

Attacks Are Staggeringly Expensive

- “Cybercrime proceeds in 2004 were $105 billion, greater than those of illegal drug sales” --- Valerie McNiven
- “Identity fraud reached $52.6 billion in 2004.” --- Javelin Strategy & Research
- “Dealing with viruses, spyware, PC theft, and other computer-related crimes costs U.S. businesses a staggering $67.2 billion a year --- FBI
- “Over 130 major intrusions exposed more than 55 million Americans to the growing variety of fraud as personal data like Social Security and credit card numbers were left unprotected” --- USA Today

The Changing Threats to Computer Security

- Vulnerable programs
  - Coding bugs, buffer overflows, parsing errors
- Malicious programs
  - Spyware, trojans, rootkits
- Misconfigured programs
  - Security features not turned on
  - Complex configuration
- Social engineering
  - Phishing/pharming

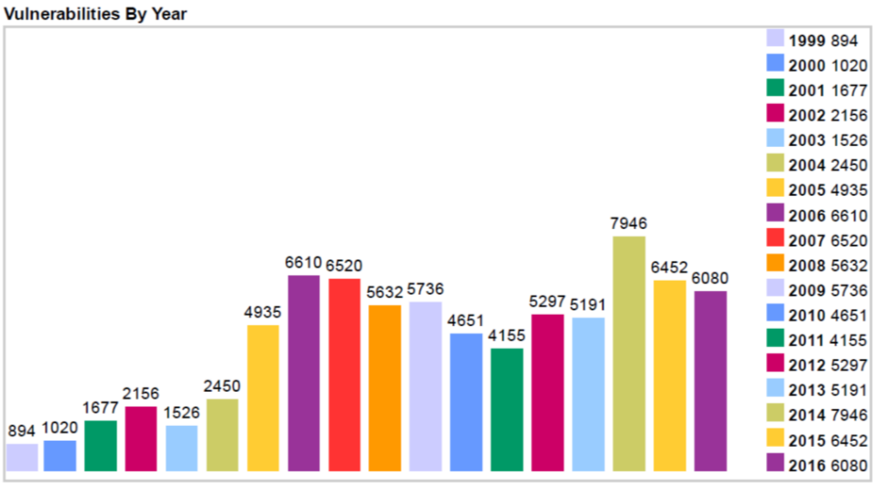
Causes

- Complexity
  - One security-related bug per thousand lines of source code
- Homogeneity
  - Same operating systems, software, libraries and hardware
- Connectivity
  - Everything is connected in the Internet
- Fundamental OS design flaws
  - Monolithic design
  - Insufficient access control

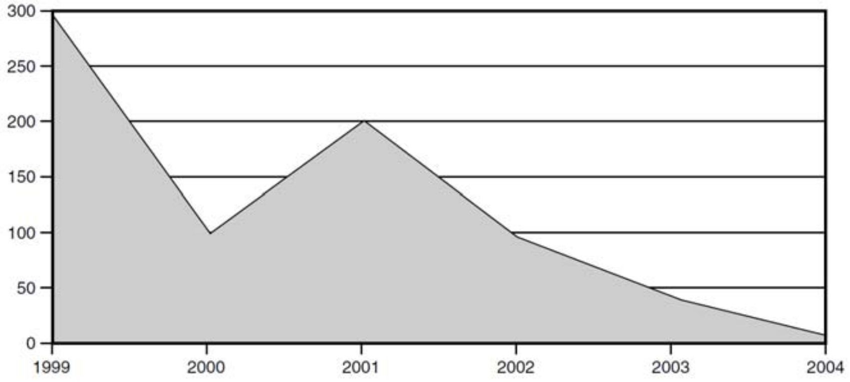
Software Security

- Common vulnerabilities:
  - Buffer overflow
  - Dangling pointer
  - Format string bugs
  - Time-of-check-to-time-of-use bugs
  - Symbolic link races
  - SQL injection
  - Directory traversal
  - Cross-site scripting
  - Cross-site request forgery
  - …

Vulnerabilities discovered per year (CERT)



Days from patch to exploit (information security, July 2004)



Software vulnerabilities in C/C++ programs

- String
- Integer
- Formatted IO
- Race Condition

Strings

- Strings—such as command-line arguments, environment variables, and console input—are of special concern in secure programming because they comprise most of the data exchanged between an end user and a software system. Graphic and Web-based applications make extensive use of text input fields and, because of standards like XML, data exchanged between programs is increasingly in string form as well. As a result, weaknesses in string representation, string management, and string manipulation have led to a broad range of software vulnerabilities and exploits.

Examples

```
int main(void) {
char Password[80];
puts("Enter 8 character password:");
gets(Password); ...
}
```

Reading unbounded stream from standard input

```
int main(int argc, char *argv[]) {
char name[2048];
strcpy(name, argv[1]);
strcat(name, " = ");
strcat(name, argv[2]); ...
}
```

Unbounded string copy and concatenation

```
#include <iostream>
int main(void) {
char buf[12];
cin >> buf;
cout << "echo: " << buf << endl;
}
```

Extracting characters from *cin* into a character array

Preparation

- echo 0 > proc/sys/kernel/randomize_va_space
- gcc –fno-stack-protector example01.c –o example-01

Problem 1

- Craft a malicious input to bypass the authentication:
  - Print “access granted” instead of “access denied”

Problem 2

- Inject arbitrary code to execute
  - A shell code template is given
  - Make a working exploit that runs “ps”

Problem 3

- Return into an existing function in libc
  - Make a working exploit that runs “ps”

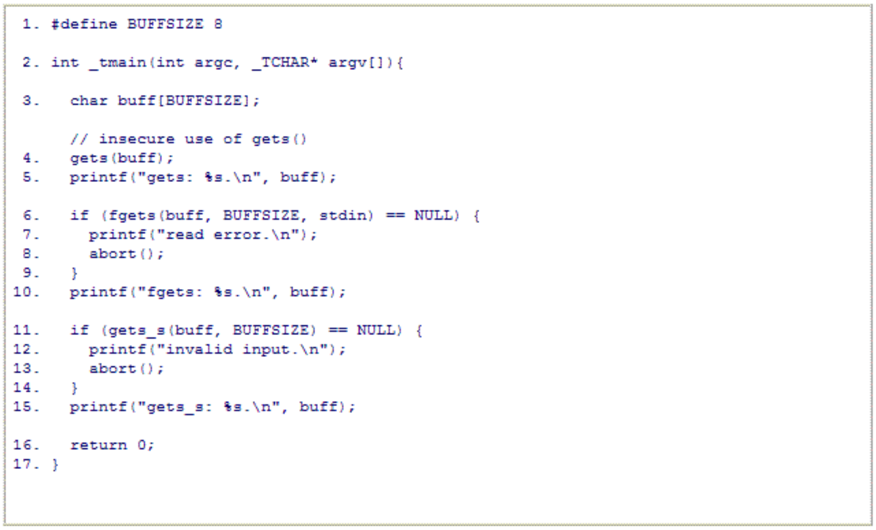
Mitigations

- Secure Coding practice
- Compiler Enhancement
- OS/Hardware Enhancement

Secure Coding: Input Validation

```
int myfunc(const char *arg) {
    char buff[100];
    if (strlen(arg) >= sizeof(buff)) {
        abort();
    }
}
```

Secure Coding: gets vs. fgets vs. gets_s



```
#define BUFFSIZE 8

int _tmain(int argc, _TCHAR* argv[]){

  char buff[BUFFSIZE];

  // insecure use of gets()
  gets(buff);
  printf("gets: %s.\n", buff);

  if (fgets(buff, BUFFSIZE, stdin) == NULL) {
    printf("read error.\n");
    abort();
  }
  printf("fgets: %s.\n", buff);

  if (gets_s(buff, BUFFSIZE) == NULL) {
    printf("invalid input.\n");
    abort();
  }
  printf("gets_s: %s.\n", buff);

  return 0;
}
```

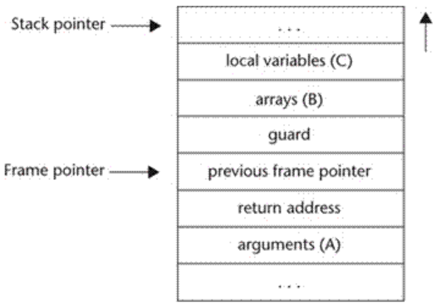
Secure Coding: strcpy & strcat

- Standard: strncpy, strncat

  strncpy(dest, source, dest_size - 1);
  dest[dest_size - 1] = '\0';

- Nonstandard: strcpy_s, strcat_s, strlcpy, strlcat

Compiler Enhancement: Canary



- Location (A) has no array or pointer variables.
- Location (B) has arrays or structures that contain arrays.
- Location (C) has no arrays.

No-Execute Protection

- In new hardware:
  - NX (No-Execute) by AMD
  - XD (eXecute-Disabled) by Intel
  - DEP (Data Execution Prevention) by Microsoft
  - A bit in the page table entry indicates if this page can be executed.
- Software emulation: PaX, W^X
- Prevent code injection attack

OS Enhancement: ASLR

- ASLR: Address Space Layout Randomization
  - Stack, heap, executable, library, etc
  - Executable/library need to be compiled to be PIE (e.g. position-independent executable)
  - On 32-bit architecture
    - 5-10% performance overhead
    - Not enough entropy: brute force can still succeed
  - On 64-bit architecture
    - Very low performance overhead
    - Enough entropy

Integer Vulnerabilities

- Integer Overflow
- Sign Error
- Truncation Error

Integer Overflow

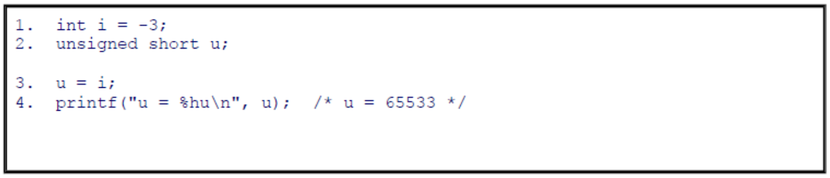
```
int i;
unsigned int j;

i = INT_MAX;  // 2,147,483,647
i++;
printf("i = %d\n", i); /* i = -2,147,483,648 */

j = UINT_MAX; // 4,294,967,295;
j++;
printf("j = %u\n", j); /* j = 0 */

i = INT_MIN; // -2,147,483,648;
i--;
printf("i = %d\n", i); /* i = 2,147,483,647 */

j = 0;
j--;
printf("j = %u\n", j); /* j = 4,294,967,295 */
```

Integer Overflow Vulnerability

```
void getComment(unsigned int len, char *src) {
  unsigned int size;

  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}

int _tmain(int argc, _TCHAR* argv[]) {
  getComment(1, "Comment ");
  return 0;
}
```

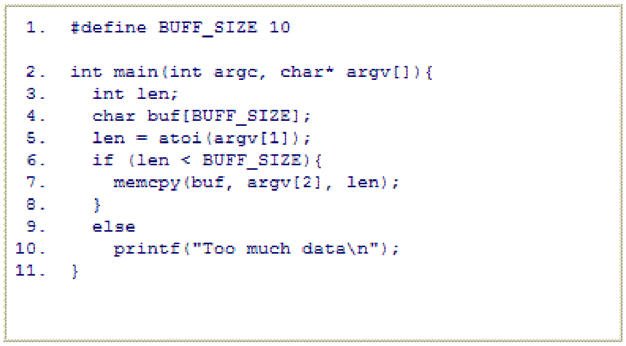
A realworld vulnerability in handling comments in JPEG files

Sign Error



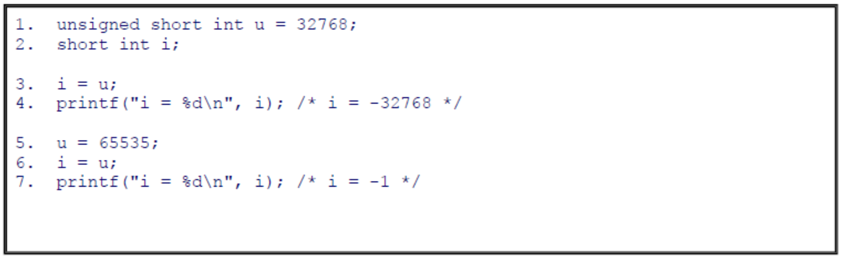
```
int i = -3;
unsigned short u;

u = i;
printf("u = %hu\n", u);  /* u = 65533 */
```

Sign Error Vulnerability



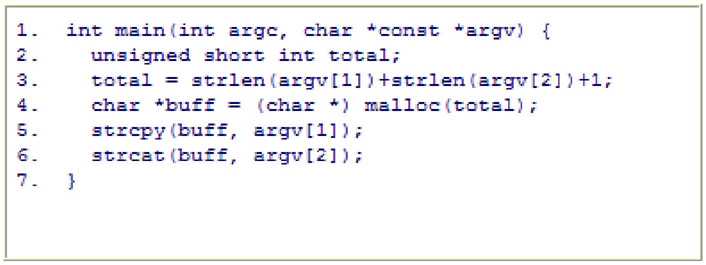
```
#define BUFF_SIZE 10

int main(int argc, char* argv[]){
  int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if (len < BUFF_SIZE){
    memcpy(buf, argv[2], len);
  }
  else
    printf("Too much data\n");
}
```

Truncation Errors



```
unsigned short int u = 32768;
short int i;

i = u;
printf("i = %d\n", i); /* i = -32768 */

u = 65535;
i = u;
printf("i = %d\n", i); /* i = -1 */
```

Truncation Error Vulnerability



```
int main(int argc, char *const *argv) {
  unsigned short int total;
  total = strlen(argv[1])+strlen(argv[2])+1;
  char *buff = (char *) malloc(total);
  strcpy(buff, argv[1]);
  strcat(buff, argv[2]);
}
```

Mitigations for Integer Vulnerabilities

- Type range checking
  - In Pascal & Ada: type day is new INTEGER range 1..31
  - In C: we need to explicitly check at runtime
- Compiler checking
  - Warning for “possible loss of data”
  - Runtime checks
    - VC++: /RTCc   GCC: -ftrapv
    - Performance overhead is high, only good for debugging
- Safe library: SafeInt
- Research Ideas
  - Static Binary Analysis
  - Dynamic Testing

Format String Vulnerabilities

- Buffer Overflow
- Read Memory Content
- Write Memory Content

Format String: Buffer Overflow

```
char buffer[512];
sprintf(buffer, "Wrong command: %s\n", user);
```

When user is too large

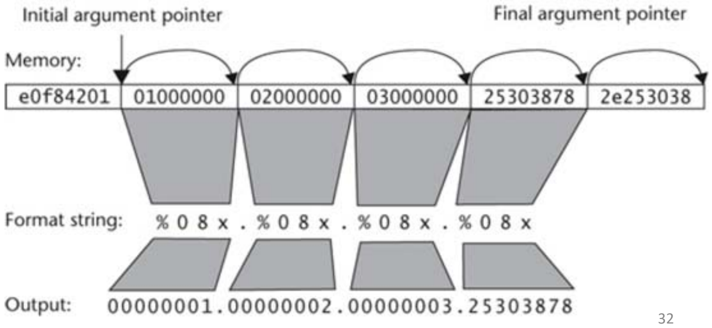
```
char outbuf[512];
char buffer[512];
sprintf(
    buffer,
    "ERR Wrong command: %.400s",
    user
);
sprintf(outbuf, buffer);
```

user = `%497d\x3c\xd3\xff\xbf<nops><shellcode>`

Format String: View Stack Content

```
char format [32];
strcpy(format, "%08x.%08x.%08x.%08x");

printf(format, 1, 2, 3);

push 3
push 2
push 1
push offset format
call _printf
add  esp,10h
```

How to view arbitrary memory content?



Format String: Write Arbitrary Memory

```
int i;
printf("hello%n\n", (int *)&i);
```

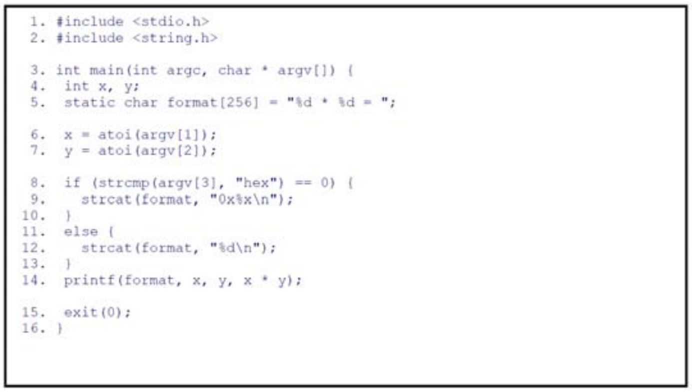
After printf, i=5

A malicious case:

```
printf("\xdc\xf5\x42\x01%08x%08x%08x%n");
```

Mitigations

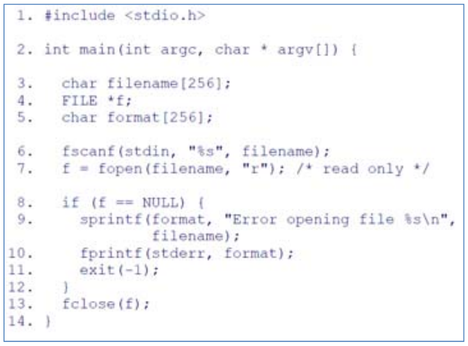
- Making format string static/constant
- Dynamic use of static content



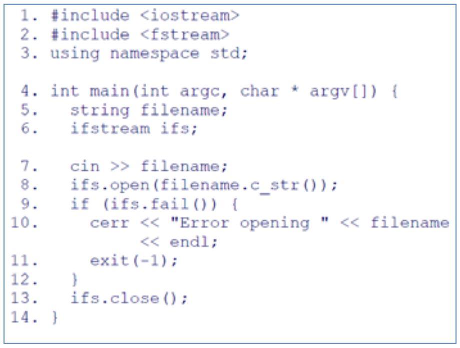
```
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[]) {
 int x, y;
 static char format[256] = "%d * %d = ";

 x = atoi(argv[1]);
 y = atoi(argv[2]);

 if (strcmp(argv[3], "hex") == 0) {
   strcat(format, "0x%x\n");
 }
 else {
   strcat(format, "%d\n");
 }
 printf(format, x, y, x * y);

 exit(0);
}
```

- snprintf versus sprintf

stdio vs. iostream



```c
#include <stdio.h>

int main(int argc, char * argv[]) {

  char filename[256];
  FILE *f;
  char format[256];

  fscanf(stdin, "%s", filename);
  f = fopen(filename, "r"); /* read only */

  if (f == NULL) {
    sprintf(format, "Error opening file %s\n",
            filename);
    fprintf(stderr, format);
    exit(-1);
  }
  fclose(f);
}
```



```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char * argv[]) {
  string filename;
  ifstream ifs;

  cin >> filename;
  ifs.open(filename.c_str());
  if (ifs.fail()) {
    cerr << "Error opening " << filename
         << endl;
    exit(-1);
  }
  ifs.close();
}
```

Mitigations (cont’d)

- Compiler checks
  - GNU C compiler flags include -Wformat, -Wformat-nonliteral, and -Wformat-security
- Research Ideas:
  - Static taint analysis
  - Dynamic taint analysis

Race Condition

- Race Condition:
  - An unanticipated execution ordering of concurrent flows that results in undesired behavior
- Three Properties:
  - Concurrency
  - Shared Object
  - Change State
- TOCTOU Race Condition
  - Time of check, time of use

Exploiting Symbolic Links

```
 1. if (stat("/some_dir/some_file", &statbuf) == -1) {
 2.   err(1, "stat");
 3. }
 4. if (statbuf.st_size >= MAX_FILE_SIZE) {
 5.   err(2, "file size");
 6. }
 7.
 8. if ((fd=open("/some_dir/some_file", O_RDONLY)) == -1) {
 9.   err(3, "open - /some_dir/some_file");
10. }
11. // process file
```

An attacker that has appropriate permission could exploit this vulnerability by executing the following commands during the race window (between lines 1 and 8):

```
rm /some_dir/some_file
ln -s attacker_file /some_dir/some_file
```

Exploiting Temporary Files

```
int fd = open("/tmp/some_file",
  O_WRONLY | O_CREAT | O_TRUNC, 0600);
```

- If a /tmp/some_file file already exists, then that file is opened and truncated.
- If /tmp/some_file is a symbolic link, then the target file referenced by the link is truncated.

```
int fd = open("/tmp/some_file",
  O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600);
```

- This call to open fails whenever /tmp/some_file already exists, including when it is a symbolic link.
- The test for file existence and the file creation are guaranteed to be atomic

Mitigation

- No easy solution
- Use file descriptor instead of filename
  - fchown vs. chown, fstat vs. stat, fchmod vs. chmod
  - Use caution with link, unlink, symlink, mkdir, rmdir, mount, unmount, etc.
- Avoid shared objects, if possible
- Least privilege
- Temporary files
  - Never reuse filenames
  - Randomize filename generation
  - Use mkstemp, rather than mktemp, tempnam, or tempnam_s

Demo --- Exploit String Vulnerability

```
int IsPasswordOkay(void) {
  char Password[12];
  gets(Password);
  if (!strcmp(Password, "goodpass"))
    return(true);
  else return(false);
}

void main(void) {
  int PwStatus;
 puts("Enter password:");
 PwStatus = IsPasswordOkay();
 if (PwStatus == false) {
   puts("Access denied");
   exit(-1);
 }
   else puts("Access granted");
 }
```

Introduction of Malware

Outline

- Malware Taxonomy & Overview
- Code Obfuscation
- Rootkit Techniques
- New Trends

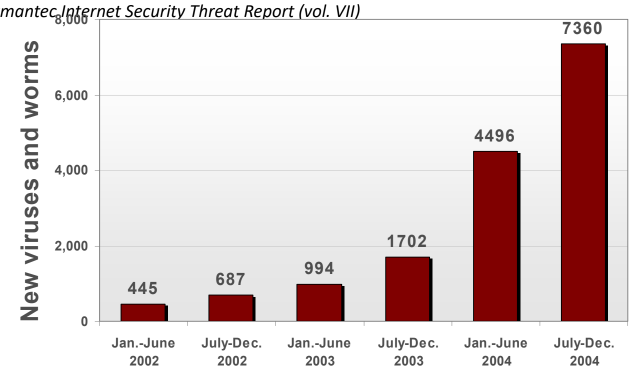
Malware Taxonomy

- Virus vs. Worm
  - Propagate itself or human involved
- Adware/Spyware
- Keylogger
- Password thief
- Network sniffer
- Mass mailer
- Backdoor
- Bot
- Driveby-download
  - Exploit browser vulnerabilities
- Rootkit

Malicious Code Problem

Malware is everywhere.

Source: Symantec Internet Security Threat Report (vol. VII)



- Large malware families.

Obfuscation Techniques

- Metamorphism
  - Upon replication, the malware generates a new (equivalent) version of itself
- Polymorphism
  - The malware encrypts its malicious payload, to be decrypted for execution
  - The encryptor and decriptor functions mutate with each replication
- Emulation
  - The malicious payload is converted into a virtual instruction set
  - An interpreter is imbedded in the malware to emulate each virtual instruction at runtime

Metamorphism

- Code Transposition (changing order of instructions)
  - Version 1 and 2 are semantically equivalent:

Version 1:	**Version 2:**
mov eax, ebx	mov ecx, 5
mov ecx, 5	mov eax, ebx
jmp +14	jmp +14
…	…

Metamorphism 2

- “nop” insertion
  - Version 1 and 2 are semantically equivalent:

Version 1:	**Version 2:**
mov eax, ebx	mov eax, ebx
mov ecx, 5	mov eax, eax
call [ebp]	test eax, eax
	nop
	inc eax
	dec eax
	mov ecx, 5
	call [ebp]
…	…

Metamorphism 3

- Register re-assignment
  - Version 1 and 2 are semantically equivalent, calling function at 0x2020 with parameter ‘5’ and clearing both ebx and eax:

Version 1:

```
mov eax, 5
push eax
call 0x2020
xor eax, eax
xor ebx, ebx
…
```

Version 2:

```
mov ebx, 5
push ebx
call 0x2020
xor ebx, ebx
xor eax, eax
…
```

Metamorphism 4

- Substitution of equivalent instruction sequences
  - Version 1 and 2 are semantically equivalent:

Version 1:

```
mov eax, 5
shl eax, 1
…
```

Version 2:

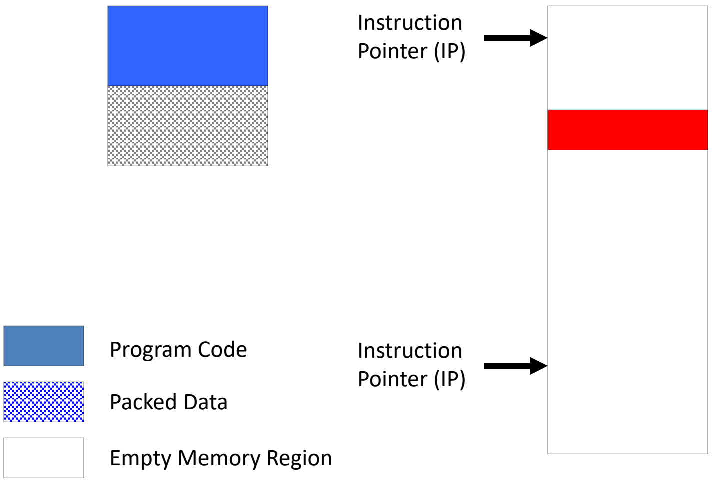
```
mov eax, 5
mul eax, eax, 2
…
```

Metamorphism 5

- Modifying condition jumps
  - Version 1, 2, and 3 are semantically equivalent:

Version 1:	**Version 2:**	**Version 3:**
mov eax, 5	mov eax, 5	mov eax, 5
test eax, eax	push 0x2020	jmp 0x2020
jnz 0x2020	ret	…
…	…	

Polymorphism (Packed Executable)



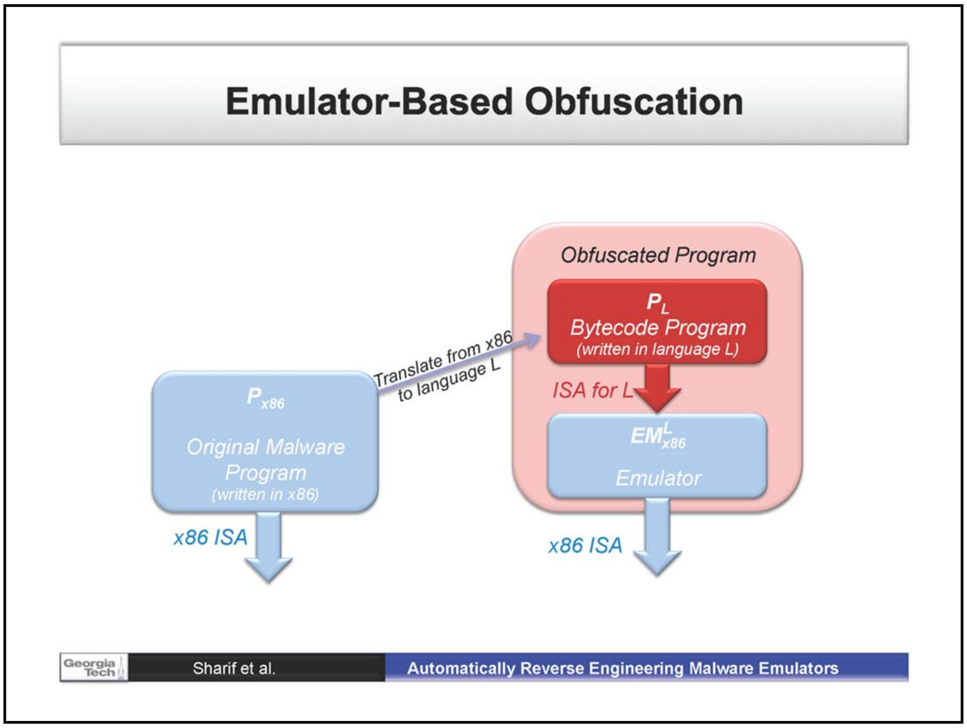
Instruction Pointer (IP)

Instruction Pointer (IP)

🟦 Program Code

▨ Packed Data

☐ Empty Memory Region



Emulator-Based Obfuscation

Obfuscated Program

P_L
Bytecode Program
(written in language L)

ISA for L

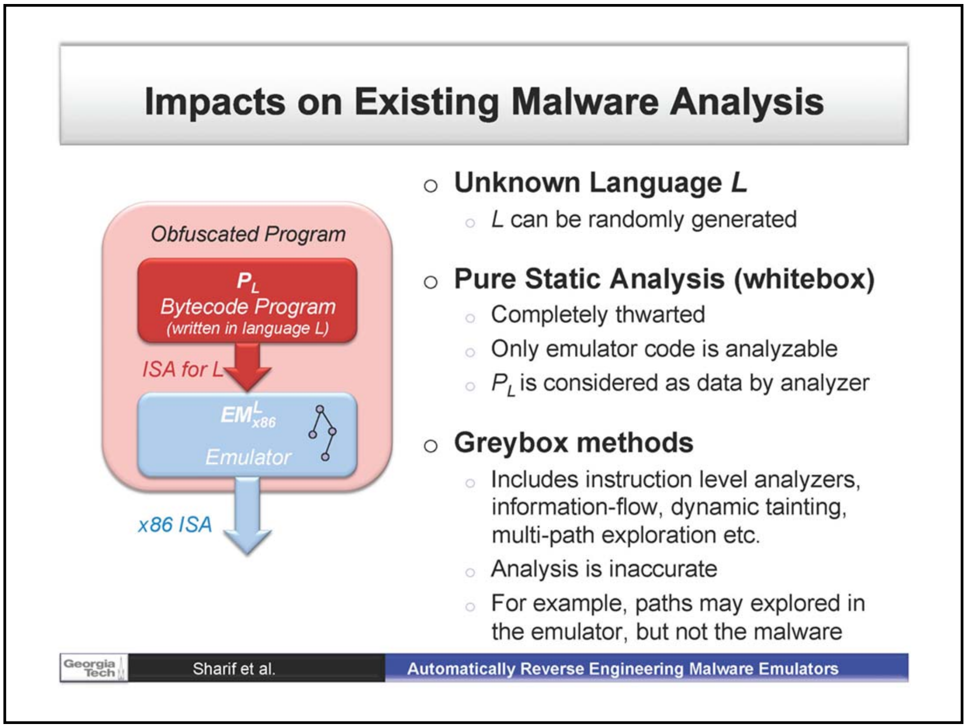
EM^L_{x86}
Emulator

x86 ISA

Translate from x86
to language L

P_{x86}
Original Malware Program
(written in x86)

x86 ISA



Impacts on Existing Malware Analysis

Obfuscated Program

P_L
Bytecode Program
(written in language L)

ISA for L

EM^L_{x86}
Emulator

x86 ISA

- **Unknown Language *L***
  - *L* can be randomly generated
- **Pure Static Analysis (whitebox)**
  - Completely thwarted
  - Only emulator code is analyzable
  - P_L is considered as data by analyzer
- **Greybox methods**
  - Includes instruction level analyzers, information-flow, dynamic tainting, multi-path exploration etc.
  - Analysis is inaccurate
  - For example, paths may explored in the emulator, but not the malware

Rootkits

- Replace system utility tools
  - E.g., ls, ps, netstat
- Hooking user-level APIs
  - Hot patching
  - Modify IAT, EAT
- Kernel hooking
  - System call table, IDT
  - Function pointers on heap (stealthier)
- Direct Kernel Object Manipulation (DKOM)
  - Unlike a process object from the active process list
  - Set pid to 0
- Virtual Machine Monitor based rootkit
  - Using hardware virtualization technology
  - Bluepill
- BIOS, Firmware rootkit …

Trend for Attackers

- From Virus to Worm to Driveby Downloads
  - No exploit -> simple exploits -> complex exploits
- From user to kernel to even lower level
  - It become harder to detect and has higher privilege
- Code obfuscation is common practice
  - Metamorphism, Polymorphism, Built-in emulator
- From hobby to profit driven
  - Economy chain
  - E.g, exploits infrastructure, botnet, black market

Trend for Defenders

- Traditional malware detection is failing
  - Signature checking: byte sequence, regular expression
  - Semantic-aware: too expensive, not practice
  - Whitelisting is a promising approach
- More security mechanism should be implemented in OS
  - Finer-grained access control
  - E.g., UAC
  - More kernel code integrity protection