Memory safety, continued

With material from Mike Hicks and Dave Levin

Today’s agenda

• gdb tutorial

• Other memory exploits

• Defenses and the continuing arms race
  • Start today, continue on Thursday
Recall
Stack and functions: Summary
Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump** to the function’s address
Recall

Stack and functions: Summary

**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump** to the function’s address

**Called function:**

4. **Push the old frame pointer** onto the stack: %ebp
5. **Set frame pointer** to where the end of the stack is right now: %ebp = %esp
6. **Push local variables** onto the stack
**Calling function:**

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump** to the function’s address

**Called function:**

4. **Push the old frame pointer** onto the stack: `%ebp`
5. **Set frame pointer** to where the end of the stack is right now: `%ebp = %esp`
6. **Push local variables** onto the stack

**Returning from function:**

7. **Reset the previous stack frame**: `%esp = %ebp, pop %ebp`
8. **Jump back** to return address: `pop %eip`
gdb tutorial

LIVE DEMO

http://original.livestream.com/filestore/logos/9aa0a959-2380-7191-77e8-06fdaf
Some gdb commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i f</td>
<td>Show info about the current frame (prev. frame, locals/args, %ebp/%eip)</td>
</tr>
<tr>
<td>i r</td>
<td>Show info about registers (%eip, %ebp, %esp, etc.)</td>
</tr>
<tr>
<td>x/&lt;n&gt; &lt;addr&gt;</td>
<td>Examine &lt;n&gt; bytes of memory starting at address &lt;addr&gt;</td>
</tr>
<tr>
<td>b &lt;function&gt; s</td>
<td>Set a breakpoint at &lt;function&gt; step through execution (into calls)</td>
</tr>
</tbody>
</table>
- run, breakpoint, step, next, continue
  - stepi, nexti
- print, x
- backtrace, info [frame, registers, args, locals]
- list, disas
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8  0x000001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbfffff8e8 0x000001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbfffff918 0x0000001f46 0x00000000 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x0000001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbfffff948 0x00000002 0x00000000
0xbffff918: 0xbfffff940 0x9be446d9 0x00000002 0xbfffff948

Saved instruction pointer  Argument
Saved base pointer  Local
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbffff918 0x00000001f46 0x00000000 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x000001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbffff948 0x00000000 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x00000002 0xbfffff948

Saved instruction pointer Argument
Saved base pointer Local
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbffff9cc
0xbffff8c8: 0xbffff8e8 0x00000001 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbffff918 0x000001f46 0x00000000 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x000001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbffff948 0x000000002 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x000000002 0xbffff948

Saved base pointer  Argument
Saved instruction pointer  Local
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbfffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbfffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbfffff918 0x000001f46 0x00000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00000000 0xa55aa55a
0xbffff908: 0x00000003 0xbfffff948 0x00000002 0x00000000
0xbffff918: 0xbfffff940 0x9be446d9 0x000000002 0xbfffff948

Saved instruction pointer  Argument
Saved base pointer     Local
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbffff9cc
0xbffff8c8: 0xbffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbffff918 0x00001f46 0x00000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbffff948 0x00000002 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x00000002 0xbffff948

Saved instruction pointer
Saved base pointer
Argument
Local

input
argc
argv
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbfffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbfffff918 0x000001f46 0x000000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbfffff948 0x00000002 0x00000000
0xbffff918: 0xbfffff940 0x9be446d9 0x000000002 0xbfffff948

Saved base pointer
Saved instruction pointer
Argument
Argument
Local
Local
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbffff918 0x00001f46 0x00000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbffff948 0x000000002 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x000000002 0xbfffff948

Saved base pointer
Saved instruction pointer
Argument
Local

answer (before)
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbfffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbfffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbfffff918 0x00001f46 0x00000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00001f0c 0xa55aa55a
0xbffff908: 0x00000000 0x00000000 0xbffff948 0x00000002 0x00000000
0xbffff918: 0xbfffff940 0x9be446d9 0x00000002 0xbfffff948

Saved instruction pointer  Argument
Saved base pointer        Local
input  argc  argv  answer (before)
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbffff9cc
0xbffff8c8: 0xbffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbffff918 0x00001f46 0x00000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbffff948 0x00000000 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x00000002 0xbffff948

Saved base pointer
Saved instruction pointer
Argument
Local
Answer (before)
inside factorial(1)

(gdb) x/32xw $ebp
0xbffff8a8: 0xbfffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbfffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbfffff918 0x00000003 0xbfffff948 0x00000002 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x0000000f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbfffff948 0x000000002 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x00000002 0xbfffff948

Saved instruction pointer  Argument
Saved base pointer  Local

Answer (before)
Other memory exploits
Heap overflow

• Stack smashing overflows a stack-allocated buffer

• You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**
  • What data gets overwritten?
Heap overflow

typedef struct _vulnerable_struct {
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two) {
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
Heap overflow

typedef struct _vulnerable_struct {
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
Heap overflow

typedef struct _vulnerable_struct {
    char buff[MAX_LEN];
    int (*cmp)(char*,char*)
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
    strcpy( s->buff, one );  // copy one into buff
    strcat( s->buff, two );  // copy two into buff
    return s->cmp( s->buff, "file://foobar" );
}
Heap overflow

```c
typedef struct _vulnerable_struct {
    char buff[MAX_LEN];
    int (*cmp)(char*, char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two) {
    strcpy(s->buff, one);
    strcat(s->buff, two);
    return s->cmp(s->buff, "file://foobar");
}
```

must have `strlen(one) + strlen(two) < MAX_LEN`

**or we overwrite** `s->cmp`
Heap overflow variants
Heap overflow variants

• Overflow into the C++ object vtable
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object’s methods
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vttable*, which contains pointers to the object’s methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object’s methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work

- **Overflow into adjacent objects**
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object’s methods
  - This table is analogous to \texttt{s->cmp} in our previous example, and a similar sort of attack will work

- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object’s methods.
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work.

- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap.

- **Overflow heap metadata**
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object’s methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work

- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap

- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object’s methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work

- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap

- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
  - Flow into that header to corrupt the heap itself
Heap overflow variants

- **Overflow into the C++ object vtable**
  - C++ objects (that contain virtual functions) are represented using a vtable, which contains pointers to the object’s methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work

- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap

- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
  - Flow into that header to corrupt the heap itself
    - Malloc implementation to do your dirty work for you!
void vulnerable()
{
    char *response;
    int nresp = packet_get_int();
    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
}
Integer overflow

```c
void vulnerable()
{
    char *response;
    int nresp = packet_get_int();
    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
}
```
void vulnerable()
{
    char *response;
    int nresp = packet_get_int();
    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
}

• What if we set nresp = 1,073,741,824?
• Assume sizeof(char*) = 4
• How many bytes are malloc’d?
Integer overflow

```c
void vulnerable()
{
    char *response;
    int nresp = packet_get_int();
    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
}
```

- What if we set `nresp = 1,073,741,824`?
- Assume `sizeof(char*) = 4`
- How many bytes are `malloc`'d?

- The `for` loop now creates an overflow!
Corrupting data

- Attacks so far primarily affect **code**
  - *Return addresses and function pointers*
Corrupting data

• Attacks so far primarily affect code
  • Return addresses and function pointers

• But attackers can overflow data as well, to
Corrupting data

- Attacks so far primarily affect code
  - Return addresses and function pointers

- But attackers can overflow data as well, to
  - Modify a secret key to be one known to the attacker, to be able to decrypt future intercepted messages
Corrupting data

• Attacks so far primarily affect code
  • Return addresses and function pointers

• But attackers can overflow data as well, to
  • Modify a secret key to be one known to the attacker, to be able to decrypt future intercepted messages
  • Modify state variables to bypass authorization checks (earlier example with authenticated flag)
Corrupting data

- Attacks so far primarily affect **code**
  - Return addresses and function pointers

- But attackers can overflow **data** as well, to
  - **Modify a secret key** to be one known to the attacker, to be able to decrypt future intercepted messages
  - **Modify state variables** to bypass authorization checks (earlier example with authenticated flag)
  - **Modify interpreted strings** used as part of commands
Corrupting data

• Attacks so far primarily affect **code**
  - Return addresses and function pointers

• But attackers can overflow **data** as well, to
  - **Modify a secret key** to be one known to the attacker, to be able to decrypt future intercepted messages
  - **Modify state variables** to bypass authorization checks (earlier example with authenticated flag)
  - **Modify interpreted strings** used as part of commands
    - E.g., to facilitate SQL injection, discussed later in the course
Read overflow

• Rather than permitting writing past the end of a buffer, a bug could permit *reading past the end*

• Might **leak secret information**
int main() {
  char buf[100], *p;

  while (1) {
    p = fgets(buf,sizeof(buf),stdin);
    len = atoi(p);
    p = fgets(buf,sizeof(buf),stdin);
    for (i=0; i<len; i++) {
      if (!iscntrl(buf[i]))
        putchar(buf[i]);
      else putchar('.');
    }
    printf("\n");
  }
}
int main() {
    char buf[100], *p;

    while (1) {
        p = fgets(buf,sizeof(buf),stdin);
        len = atoi(p);
        for (i=0; i<len; i++) {
            if (!iscntrl(buf[i]))
                putchar(buf[i]);
            else putchar(‘.’);
        }
        printf(“\n”);
    }
    ...
int main() {
    char buf[100], *p;

    while (1) {
        p = fgets(buf,sizeof(buf),stdin);
        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        for (i=0; i<len; i++) {
            if (!iscntrl(buf[i]))
                putchar(buf[i]);
            else putchar(' .');
        }
        printf("\n");
    }
    ...
}
int main() {
    char buf[100], *p;

    while (1) {
        p = fgets(buf,sizeof(buf),stdin);
        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        for (i=0; i<len; i++) {
            if (!iscntrl(buf[i]))
                putchar(buf[i]);
            else putchar('.');
        }
        printf("\n");
    }
    ...
}
int main() {
    char buf[100], *p;

    while (1) {
        p = fgets(buf,sizeof(buf),stdin);
        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        for (i=0; i<len; i++) {
            if (!iscntrl(buf[i]))
                putchar(buf[i]);
            else putchar('.');
        }
        printf("\n");
    }
    ...
}

Read overflow

Read integer

Read message

Echo back (partial) message

len may exceed actual message length!
Sample transcript
Sample transcript

% ./echo-server
Sample transcript

% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
Sample transcript

% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|

}{ OK: input length < buffer size
Sample transcript

```bash
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

\[ \text{OK: input length} \quad < \text{buffer size} \]

\[ \text{BAD: length} > \text{size} ! \]
Sample transcript

% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|

{OK: input length < buffer size}

{BAD: length > size !}

leaked data
Heartbleed, again

**SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).**

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "CokeBaSt".

HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "CokeBaSt".
Stale memory
Stale memory

- A **dangling pointer bug** occurs when a pointer is freed, but the program continues to use it
Stale memory

• A **dangling pointer bug** occurs when a pointer is freed, but the program continues to use it.

• An attacker can **arrange for the freed memory to be reallocated** and under his control.
  • When the dangling pointer is dereferenced, it will access attacker-controlled data.
Stale memory

```
struct foo { int (*cmp)(char*, char*); };
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data.
Stale memory

```c
struct foo { int (*cmp)(char*, char*); };
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data
Stale memory

```c
struct foo { int (*cmp)(char*, char*); };
struct foo *p = malloc(...);
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data
Stale memory

struct foo { int (*cmp)(char*,char*); };  
struct foo *p = malloc(...);  
free(p);

• When the dangling pointer is dereferenced, it will access attacker-controlled data
Stale memory

struct foo {
  int (*cmp)(char*, char*);
};

struct foo *p = malloc(...);
free(p);

• When the dangling pointer is dereferenced, it will access attacker-controlled data
```c
struct foo { int (*cmp)(char*, char*); };
struct foo *p = malloc(...);
free(p);

q = malloc(...) //reuses memory
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data
Stale memory

```c
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(...);
free(p);
  . . .
q = malloc(...)  //reuses memory
*q = 0xdeadbeef;  //attacker control
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data
Stale memory

```c
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(...);
free(p);

q = malloc(...) //reuses memory
*q = 0xdeadbeef; //attacker control
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data
Stale memory

```
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(...);
free(p);

q = malloc(...)  //reuses memory
*q = 0xdeadbeef;  //attacker control

p->cmp("hello","hello");  //dangling ptr
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data.
IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

**Pointing to the Flaw**

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.
The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.
Format string vulnerabilities
void print_record(int age, char *name)
{
    printf("Name: %s\tAge: %d\n", name, age);
}
Formatted I/O

• Recall: C’s `printf` family of functions

```c
void print_record(int age, char *name)
{
    printf("Name: %s\tAge: %d\n", name, age);
}
```
Formatted I/O

• Recall: C’s `printf` family of functions
• Format specifiers, list of arguments

```c
void print_record(int age, char *name)
{
    printf("Name: %s\tAge: %d\n", name, age);
}
```
Formatted I/O

• Recall: C’s `printf` family of functions

• Format specifiers, list of arguments
  • Specifier indicates type of argument (%s, %i, etc.)

```c
void print_record(int age, char *name)
{
    printf(“Name: %s\tAge: %d\n”, name, age);
}
```
Formatted I/O

• Recall: C’s `printf` family of functions

• Format specifiers, list of arguments
  • Specifier indicates type of argument (%s, %i, etc.)
  • Position in string indicates argument to print

```c
void print_record(int age, char *name)
{
    printf("Name: %s\tAge: %d\n", name, age);
}
```
What’s the difference?
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}

void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s", buf);
}
What’s the difference?

void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf); /* Attacker controls the format string */
}

void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
printf implementation

```c
int i = 10;
printf("%d %p\n", i, &i);
```
printf implementation

```c
int i = 10;
printf("%d %p\n", i, &i);
```

```
0x00000000 0xffffffff
...	%ebp  %eip  &fmt  10  &i
```
printf implementation

```c
int i = 10;
printf("%d %p\n", i, &i);
```

printf’s stack frame
printf implementation

```c
int i = 10;
printf("%d %p\n", i, &i);
```

`printf’s stack frame`

`caller’s stack frame`
printf implementation

```c
int i = 10;
printf("%d %p\n", i, &i);
```

printf's stack frame

caller's stack frame
printf implementation

int i = 10;
printf("%d %p\n", i, &i);

• printf takes a variable number of arguments
• Doesn’t know where the stack frame “ends”
• Keeps reading from stack until out of format specifiers
printf implementation

int i = 10;
printf("%d %p
", i, &i);

• printf takes a variable number of arguments
• Doesn’t know where the stack frame “ends”
• Keeps reading from stack until out of format specifiers
printf implementation

int i = 10;
printf("%d %p\n", i, &i);

• printf takes a variable number of arguments
• Doesn’t know where the stack frame “ends”
• Keeps reading from stack until out of format specifiers
printf implementation

- printf takes a variable number of arguments
- Doesn’t know where the stack frame “ends”
- Keeps reading from stack until out of format specifiers
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}

"\%d  \%x"
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}

"%d %x"

caller's stack frame

0x0000000000 0xffffffff

... %ebp %eip &fmt
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}

"%d %x"

caller’s stack frame

0x000000000000

0xffffffffffffffff
```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

```
"%d %x"
```

```
0x00000000 | caller's stack frame
```

```
0xffffffff | 0xffffffff
```

```
... %ebp %eip &fmt
```

```
```
Format string vulnerabilities
Format string vulnerabilities

- `printf("100% dinosaur");`
Format string vulnerabilities

- `printf("100% dinosaur");`
  - Prints stack entry 4 bytes above saved %eip
Format string vulnerabilities

- `printf("100% dinosaur");`
  - Prints stack entry 4 byes above saved `%eip`

- `printf("%s");`
Format string vulnerabilities

- `printf("100% dinosaur");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
  - Prints bytes \textit{pointed to} by that stack entry
Format string vulnerabilities

• `printf("100% dinosaur");`
  • Prints stack entry 4 bytes above saved %eip

• `printf("%s");`
  • Prints bytes *pointed to* by that stack entry

• `printf("%d %d %d %d ...");`
Format string vulnerabilities

• `printf("100% dinosaur");`
  • Prints stack entry 4 bytes above saved %eip

• `printf("%s");`
  • Prints bytes *pointed to* by that stack entry

• `printf("%d %d %d %d ...");`
  • Prints a series of stack entries as integers
Format string vulnerabilities

- `printf(“100% dinosaur”);`
  - Prints stack entry 4 bytes above saved %eip

- `printf(“%s”);`
  - Prints bytes pointed to by that stack entry

- `printf(“%d %d %d %d ...”);`
  - Prints a series of stack entries as integers

- `printf(“%08x %08x %08x %08x ...”);`
Format string vulnerabilities

• `printf("100% dinosaur");`
  - Prints stack entry 4 bytes above saved %eip

• `printf("%s");`
  - Prints bytes pointed to by that stack entry

• `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

• `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex
Format string vulnerabilities

- `printf("100% dinosaur");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
  - Prints bytes pointed to by that stack entry

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex

- `printf("100% not vulnerable!");`
Format string vulnerabilities

- `printf("100% dinosaur");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex

- `printf("100% not vulnerable!");`
  - *WRTIES* the number 3 to address pointed to by stack entry
Why is this a buffer overflow?

• We should think of this as a buffer overflow in the sense that
  • The stack itself can be viewed as a kind of buffer
  • Size of that buffer is determined by the number and size of the arguments passed to a function
• Providing a bogus format string thus induces the program to overflow that “buffer”
Vulnerability prevalence

100% preventable!

number of vulnerabilities that involve format string bugs

Time to switch hats
Time to switch hats

We have seen many styles of attack
Time to switch hats

We have seen many styles of attack

How can we defend against them?
Defenses
Against memory and buffer attacks

http://www.full-stop.net/wp-content/uploads/2012/05/Great-wall-of-china.jpeg
Stepping back
Stepping back

What do these attacks have in common?
Stepping back

What do these attacks have in common?

1. The attacker is able to control some data that is used by the program
Stepping back

What do these attacks have in common?

1. The attacker is able to **control some data** that is used by the program

2. The use of that data permits **unintentional access to some memory area** in the program
   - Past a buffer
   - To arbitrary positions on the stack / in the heap
Outline
Outline

• **Memory safety and type safety**
  • Properties that, if satisfied, ensure an application is immune to memory attacks
Outline

• **Memory safety and type safety**
  • Properties that, if satisfied, ensure an application is immune to memory attacks

• Automatic defenses
  • Stack canaries
  • Address space layout randomization (ASLR)
Outline

• Memory safety and type safety
  • Properties that, if satisfied, ensure an application is immune to memory attacks

• Automatic defenses
  • Stack canaries
  • Address space layout randomization (ASLR)

• Return-oriented programming (ROP) attack
  • How Control Flow Integrity (CFI) can defeat it
Outline

- **Memory safety and type safety**
  - Properties that, if satisfied, ensure an application is immune to memory attacks
- Automatic defenses
  - Stack canaries
  - Address space layout randomization (ASLR)
- Return-oriented programming (ROP) attack
  - How Control Flow Integrity (CFI) can defeat it
- Secure coding
“Once you learn, though, you’ll never forget.”

Memory Safety

https://diarrheapolice.files.wordpress.com/2015/08/elephant-bicycle-never-forget.png?w=590&h=334
What is memory safety?
What is memory safety?

A memory safe program execution:
What is memory safety?

A memory safe program execution:

1. Only creates pointers through **standard means**
What is memory safety?

A memory safe program execution:

1. Only creates pointers through **standard means**
   - \( p = \text{malloc}(\ldots) \), or \( p = &x \), or \( p = &\text{buf}[5] \), etc.
What is memory safety?

A memory safe program execution:

1. Only creates pointers through *standard means*
   - \( p = \text{malloc}(\ldots) \), or \( p = &x \), or \( p = &\text{buf}[5] \), etc.

2. Only uses a pointer to access memory that “belongs” to that pointer
What is memory safety?

A memory safe program execution:

1. Only creates pointers through **standard means**
   - \( p = \text{malloc}(...) \), or \( p = \&x \), or \( p = \&\text{buf}[5] \), etc.

2. Only uses a pointer to access memory that **“belongs” to that pointer**

Combines two ideas:
What is memory safety?

A memory safe program execution:

1. Only creates pointers through **standard means**
   - \( p = \text{malloc}(\ldots) \), or \( p = \&x \), or \( p = \&\text{buf}[5] \), etc.

2. Only uses a pointer to access memory that “belongs” to that pointer

Combines two ideas:

**temporal safety** and **spatial safety**
Spatial safety
Spatial safety

• View pointers as **capabilities**: triples \((p,b,e)\)
  • \(p\) is the actual pointer (current address)
  • \(b\) is the base of the memory region it may access
  • \(e\) is the extent (bounds) of that region (count)
Spatial safety

- View pointers as **capabilities**: triples \((p, b, e)\)
  - \(p\) is the actual pointer (current address)
  - \(b\) is the base of the memory region it may access
  - \(e\) is the extent (bounds) of that region (count)

- Access allowed \(iff\) \(b \leq p \leq (e - \text{sizeof(\text{typeof}(p))})\)
Spatial safety

- View pointers as **capabilities**: triples \((p, b, e)\)
  - \(p\) is the actual pointer (current address)
  - \(b\) is the base of the memory region it may access
  - \(e\) is the extent (bounds) of that region (count)

- **Access allowed** \(iff\) \(b \leq p \leq (e - \text{sizeof}(\text{typeof}(p)))\)

- Operations:
  - Pointer arithmetic increments \(p\), leaves \(b\) and \(e\) alone
  - Using \&: \(e\) determined by size of original type
Examples

```c
int x; // assume sizeof(int)=4
```
Examples

int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
```
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)−4
```
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)−4
*z = 3;       // Bad: &x ≤ &x+4 < ( &x+4 )−4
```
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)−4
*z = 3;       // Bad: &x ≤ &x+4 ≤ (&x+4)−4
```

```c
struct foo {
    char buf[4];
    int x;
};
```
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)−4
*z = 3;       // Bad: &x ≤ &x+4  ≠  (&x+4)−4
```

```c
struct foo f = { "cat", 5 };```

```c
struct foo {
    char buf[4];  
    int x;
};
```
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)−4
*z = 3;       // Bad: &x ≤ &x+4 ≤ (&x+4)−4
```

```c
struct foo {  
    char buf[4];
    int x;
};
```

```c
struct foo f = { "cat", 5 };  
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
```
Examples

```
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;       // Bad: &x ≤ &x+4 ≺ (&x+4)-4
```

```
struct foo {
    char buf[4];
    int x;
};

struct foo f = { "cat", 5 };
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = 's';      // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
```
Examples

```
int x;       // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;      // OK: &x ≤ &x ≤ (&x+4)−4
*z = 3;      // Bad: &x ≤ &x+4 < (&x+4)−4
```

```
struct foo f = { "cat", 5 };
char *y = &f.buf;   // p = b = &f.buf, e = &f.buf+4
y[3] = 's';        // OK: p = &f.buf+3 ≤ (&f.buf+4)−1
y[4] = 'y';        // Bad: p = &f.buf+4 < (&f.buf+4)−1
```
Visualized example

```c
struct foo {
    int x;
    int y;
    char *pc;
};
```
Visualized example

```c
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
```
Visualized example

```c
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
```
Visualized example

```c
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
```
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
```
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```
Visualized example

```c
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
ext *px = &pf->x;
```
No buffer overflows

• A buffer overflow violates spatial safety

```c
void copy(char *src, char *dst, int len)
{
    int i;
    for (i=0;i<len;i++) {
        *dst = *src;
        src++;
        dst++;
    }
}
```

• Overrunning bounds of source and/or destination buffers implies either `src` or `dst` is illegal
No format string attacks

• The call to `printf` dereferences illegal pointers

```c
char *buf = "%d %d %d\n";
printf(buf);
```

• View the stack as a buffer defined by the number and types of the arguments it provides

• The extra format specifiers construct pointers beyond the end of this buffer and dereference them

• Essentially a kind of buffer overflow
Temporal safety

- Violated when trying to access **undefined memory**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play
Temporal safety

- Violated when trying to access **undefined memory**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play

- Memory regions either **defined** or **undefined**
  - Defined means allocated (and active)
  - Undefined means unallocated, uninitialized, or deallocated
No dangling pointers
No dangling pointers

- Accessing a freed pointer violates temporal safety
No dangling pointers

- Accessing a freed pointer violates temporal safety

```c
int *p = malloc(sizeof(int));
```
No dangling pointers

- Accessing a freed pointer violates temporal safety

```c
int *p = malloc(sizeof(int));
*p = 5;
```
No dangling pointers

- Accessing a freed pointer violates temporal safety

```c
int *p = malloc(sizeof(int));
*p = 5;
free(p);
```
No dangling pointers

- Accessing a freed pointer violates temporal safety

```c
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to p.
No dangling pointers

• Accessing a freed pointer violates temporal safety

int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation

The memory dereferenced no longer belongs to p.

• Accessing uninitialized pointers is similarly not OK:
No dangling pointers

• Accessing a freed pointer violates temporal safety

```c
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to `p`.

• Accessing uninitialized pointers is similarly not OK:

```c
int *p;
```
No dangling pointers

• Accessing a freed pointer violates temporal safety

```c
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to p.

• Accessing uninitialized pointers is similarly not OK:

```c
int *p;
*p = 5; // violation
```
Integer overflows?
Integer overflows?

• Integer overflows are themselves allowed
  • But can’t become illegal pointers
Integer overflows?

- Integer overflows are themselves allowed
  - But can’t become illegal pointers
- Integer overflows often enable buffer overflows
Integer overflows?

```c
int f() {
    unsigned short x = 65535;
}
```

- Integer overflows are themselves allowed
  - But can’t become illegal pointers
- Integer overflows often enable buffer overflows
Integer overflows?

- Integer overflows are themselves allowed
- But can’t become illegal pointers
- Integer overflows often enable buffer overflows

```c
int f() {
    unsigned short x = 65535;
    x++; // overflows to become 0
}
```
Integer overflows?

```c
int f() {
    unsigned short x = 65535;
    x++; // overflows to become 0
    printf("%d\n", x); // memory safe
}
```

- Integer overflows are themselves allowed
- But can’t become illegal pointers
- Integer overflows often enable buffer overflows
Integer overflows?

int f() {
    unsigned short x = 65535;
    x++; // overflows to become 0
    printf("%d\n",x); // memory safe
    char *p = malloc(x); // size-0 buffer!
}

• Integer overflows are themselves allowed
  • But can’t become illegal pointers

• Integer overflows often enable buffer overflows
Integer overflows?

```c
int f() {
    unsigned short x = 65535;
    x++; // overflows to become 0
    printf("%d\n",x); // memory safe
    char *p = malloc(x); // size-0 buffer!
    p[1] = 'a'; // violation
}
```

- Integer overflows are themselves allowed
- But can’t become illegal pointers
- Integer overflows often enable buffer overflows
Integer overflows?

```c
int f() {
    unsigned short x = 65535;
    x++; // overflows to become 0
    printf("%d\n", x); // memory safe
    char *p = malloc(x); // size-0 buffer!
    p[1] = 'a'; // violation
}
```

• Integer overflows are themselves allowed
  • But can’t become illegal pointers

• Integer overflows often enable buffer overflows

For more on memory safety, see http://www.pl-enthusiast.net/2014/07/21/memory-safety/
How to get memory safety?

• The easiest way to avoid all of these vulnerabilities is to use a memory-safe language.
How to get memory safety?

• The easiest way to avoid all of these vulnerabilities is to use a memory-safe language

• Modern languages are memory safe
  • Java, Python, C#, Ruby
  • Haskell, Scala, Go, Objective Caml, Rust
How to get memory safety?

• The easiest way to avoid all of these vulnerabilities is to use a memory-safe language

• Modern languages are memory safe
  • Java, Python, C#, Ruby
  • Haskell, Scala, Go, Objective Caml, Rust

• In fact, these languages are type safe, which is even better (more on this shortly)
Recall

C and C++ are still very popular.

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. C</td>
<td>![Phone][Desktop][Server]</td>
<td>100.0</td>
</tr>
<tr>
<td>2. Java</td>
<td>![Network][Phone][Desktop]</td>
<td>98.1</td>
</tr>
<tr>
<td>3. Python</td>
<td>![Network][Desktop]</td>
<td>98.0</td>
</tr>
<tr>
<td>4. C++</td>
<td>![Phone][Desktop][Server]</td>
<td>95.9</td>
</tr>
<tr>
<td>5. R</td>
<td>![Desktop]</td>
<td>87.9</td>
</tr>
<tr>
<td>6. C#</td>
<td>![Network][Phone][Desktop]</td>
<td>86.7</td>
</tr>
<tr>
<td>7. PHP</td>
<td>![Network]</td>
<td>82.8</td>
</tr>
<tr>
<td>8. JavaScript</td>
<td>![Network][Phone]</td>
<td>82.2</td>
</tr>
<tr>
<td>9. Ruby</td>
<td>![Network][Desktop]</td>
<td>74.5</td>
</tr>
<tr>
<td>10. Go</td>
<td>![Network][Desktop]</td>
<td>71.9</td>
</tr>
</tbody>
</table>

[![Network][Phone][Desktop][Server]](spectrum.ieee.org/computing/software/the-2016-top-programming-languages)
Memory safety for C
Memory safety for C

- C/C++ are here to stay.
  - You *can* write memory safe programs with them
  - But the language provides no guarantee
Memory safety for C

• C/C++ are here to stay.
  • You can write memory safe programs with them
  • But the language provides no guarantee

• Compilers could add code to check for violations
  • Out-of-bounds: immediate failure (Java ArrayBoundsException)
Memory safety for C

• **C/C++ are here to stay.**
  • You can write memory safe programs with them
  • But the language provides no guarantee

• Compilers could add code to **check for violations**
  • Out-of-bounds: immediate failure (Java `ArrayOutOfBoundsException`)

• This idea has been around for more than 20 years. **Performance has been the limiting factor.**
  • Work by Jones and Kelly in 1997 adds 12x overhead
  • Valgrind memcheck adds 17x overhead
Research progress

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs

- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking, highly flexible
Research progress

• **CCured** (2004), 1.5x slowdown
  • But no checking in libraries
  • Compiler rejects many safe programs

• **Softbound/CETS** (2010): 2.16x slowdown
  • Complete checking, highly flexible

• **Intel MPX** hardware (2015 in Linux)
  • Hardware support to make checking faster

Research progress

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs
- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking, highly flexible
- **Intel MPX** hardware (2015 in Linux)
  - Hardware support to make checking faster

1942 report
american typewriter
carbon type
mom's typewriter
kingthings trypewriter
my underwood
underwood champion
sears tower
veteran typewriter

Type Safety
Type safety
Type safety

• Each object is ascribed a type (int, pointer to int, pointer to function), and
Type safety

• Each object is ascribed a **type** (int, pointer to int, pointer to function), and

• Operations on the object are always **compatible** with the object’s type
Type safety

- Each object is ascribed a type (int, pointer to int, pointer to function), and

- Operations on the object are always compatible with the object’s type
  - Type safe programs do not “go wrong” at run-time
Type safety

• Each object is ascribed a type (int, pointer to int, pointer to function), and

• Operations on the object are always compatible with the object’s type
  • Type safe programs do not “go wrong” at run-time

• Type safety is stronger than memory safety
Type safety

• Each object is ascribed a type (int, pointer to int, pointer to function), and

• Operations on the object are always compatible with the object’s type
  • Type safe programs do not “go wrong” at run-time

• **Type safety** is **stronger** than memory safety

```c
int (*cmp)(char*,char*);
```
Type safety

• Each object is ascribed a **type** (`int`, pointer to `int`, pointer to function), and

• Operations on the object are always *compatible* with the object’s type
  • Type safe programs do not “go wrong” at run-time

• **Type safety** is **stronger** than memory safety

```c
int (*cmp)(char*, char*);
int *p = (int*)malloc(sizeof(int));
```
Type safety

• Each object is ascribed a type (int, pointer to int, pointer to function), and

• Operations on the object are always compatible with the object’s type
  • Type safe programs do not “go wrong” at run-time

• Type safety is stronger than memory safety

```c
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
```
Type safety

• Each object is ascribed a type (int, pointer to int, pointer to function), and

• Operations on the object are always compatible with the object’s type
  • Type safe programs do not “go wrong” at run-time

• Type safety is stronger than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
```
Type safety

- Each object is ascribed a type (int, pointer to int, pointer to function), and

- Operations on the object are always compatible with the object’s type
  - Type safe programs do not “go wrong” at run-time

- Type safety is stronger than memory safety

```c
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
```

Memory safe, NOT type safe
Type safety

- Each object is ascribed a **type** (int, pointer to int, pointer to function), and

- Operations on the object are always *compatible* with the object’s type
  - Type safe programs do not “go wrong” at run-time

- **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*, char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*, char*))p;
cmp("hello", "bye");  // crash!
```
Aside: Dynamic Typing

- Dynamically typed languages
Aside: Dynamic Typing

- Dynamically typed languages
  - Don’t require type declaration
Aside: Dynamic Typing

• Dynamically typed languages
  • Don’t require type declaration
  • e.g., Ruby and Python
Aside: Dynamic Typing

• Dynamically typed languages
  • Don’t require type declaration
  • e.g., Ruby and Python
  • Can be viewed as type safe
Aside: Dynamic Typing

• Dynamically typed languages
  • Don’t require type declaration
  • e.g., Ruby and Python
  • Can be viewed as type safe

• Each object has one type: **Dynamic**
Aside: Dynamic Typing

• Dynamically typed languages
  • Don’t require type declaration
  • e.g., Ruby and Python
  • Can be viewed as type safe

• Each object has one type: Dynamic
  • Each operation on a Dynamic object is permitted, but may be unimplemented
Aside: Dynamic Typing

• Dynamically typed languages
  • Don’t require type declaration
  • e.g., Ruby and Python
  • Can be viewed as type safe

• Each object has one type: **Dynamic**
  • Each operation on a Dynamic object is permitted, but *may be unimplemented*
  • In this case, it *throws an exception*
Aside: Dynamic Typing

- Dynamically typed languages
  - Don’t require type declaration
  - e.g., Ruby and Python
  - Can be viewed as type safe

- Each object has one type: Dynamic
  - Each operation on a Dynamic object is permitted, but may be unimplemented
  - In this case, it throws an exception
  - Checked at runtime not compile time!
Enforce invariants
Enforce invariants

- Types most useful for enforcing invariants
- (Examples next slide)
Enforce invariants

• Types most useful for enforcing invariants
  • (Examples next slide)

• Enforcement of abstract types: modules with hidden representation
  • Allow reasoning more confidently about their isolation from the rest of the program
Enforce invariants

• Types most useful for enforcing invariants
  • (Examples next slide)

• Enforcement of abstract types: modules with hidden representation
  • Allow reasoning more confidently about their isolation from the rest of the program

For more on type safety, see http://www.pl-enthusiast.net/2014/08/05/type-safety/
Types for Security

• Use types to enforce **security property** invariants
  • Invariants about data’s privacy and integrity
  • Enforced by the type checker
Types for Security

- Use types to enforce **security property** invariants
  - Invariants about data’s privacy and integrity
  - Enforced by the type checker

- **Example**: *Java with Information Flow (JIF)*

http://www.cs.cornell.edu/jif
Types for Security

• Use types to enforce **security property** invariants
  • Invariants about data’s privacy and integrity
  • Enforced by the type checker

• **Example:** *Java with Information Flow (JIF)*

```java
int{Alice, Bob} x;
```

http://www.cs.cornell.edu/jif
Types for Security

- Use types to enforce **security property** invariants
  - Invariants about data’s privacy and integrity
  - Enforced by the type checker

- Example: **Java with Information Flow (JIF)**

```java
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
```

http://www.cs.cornell.edu/jif
Types for Security

• Use types to enforce security property invariants
  • Invariants about data’s privacy and integrity
  • Enforced by the type checker

• Example: Java with Information Flow (JIF)

```java
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
x = y; //OK: policy on x is stronger
```

http://www.cs.cornell.edu/jif
Types for Security

• Use types to enforce **security property** invariants
  • Invariants about data’s privacy and integrity
  • Enforced by the type checker

• **Example:** **Java with Information Flow (JIF)**

```java
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
x = y;  //OK: policy on x is stronger
y = x;  //BAD: policy on y is weaker
```

http://www.cs.cornell.edu/jif
Types for Security

- Use types to enforce **security property** invariants
  - Invariants about data’s privacy and integrity
  - Enforced by the type checker

- **Example:** Java with Information Flow (JIF)

```java
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is weaker
```

Types have **security labels** that govern **information flow**

http://www.cs.cornell.edu/jif
Why not type safety?
Why not type safety?

• C/C++ often chosen for performance reasons
  • Manual memory management
  • Tight control over object layouts
  • Interaction with low-level hardware
Why not type safety?

• C/C++ often chosen for performance reasons
  • Manual memory management
  • Tight control over object layouts
  • Interaction with low-level hardware

• Enforcement of type safety is typically expensive
  • Garbage collection avoids temporal violations
    • Can be as fast as malloc/free, often uses much more memory
  • Bounds and null-pointer checks avoid spatial violations
  • Hiding representation may inhibit optimization
    • Many C-style casts, pointer arithmetic, & operator, not allowed
A new hope?
A new hope?

• Many applications do not need C/C++
  • Or the risks that come with it
A new hope?

• Many applications do not need C/C++
  • Or the risks that come with it

• New languages aim to provide similar features to C/C++ while remaining type safe
  • Google’s Go, Mozilla’s Rust, Apple’s Swift
A new hope?

• Many applications do not need C/C++
  • Or the risks that come with it

• New languages aim to provide similar features to C/C++ while remaining type safe
  • Google’s Go, Mozilla’s Rust, Apple’s Swift