This time

We will continue

Buffer overflows

and other memory safety vulnerabilities

By looking at

Overflow Defenses

- Everything you've always wanted to know about `gdb` but were too afraid to ask
- Overflow defenses
- Other memory safety vulnerabilities

What’s wrong with this code?

Suppose that it has higher privilege than the user

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...                ~attacker/mystuff.txt
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

~attacker/mystuff.txt

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What’s wrong with this code?

Suppose that it has higher privilege than the user

~attacker/mystuff.txt

uid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf(“cannot access file\n”);
        exit(-1);
    }
```

ln -s /usr/sensitive ~attacker/mystuff.txt

euid

```
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf(“%s\n”, buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...                    ~attacker/mystuff.txt
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
 ln -s /usr/sensitive ~attacker/mystuff.txt
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

"Time of Check/Time of Use" Problem (TOCTOU)

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }



    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }



    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);   // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

Avoiding TOCTOU

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);    // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    seteuid(euid);   // Restore privileges
    printf(buf);
}
```

uid

euid

Memory safety attacks

- Buffer overflows
  - Can be used to read/write data on stack or heap
  - Can be used to inject code (ultimately root shell)
- Format string errors
  - Can be used to read/write stack data
- Integer overflow errors
  - Can be used to change the control flow of a program
- TOCTOU problem
  - Can be used to raise privileges

General defenses against memory-safety

Defensive coding practices

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - 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 (or even exit) than run malicious code

http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

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

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

- 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

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

Think about all potential inputs, no matter how peculiar

Secure coding practices

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

Think about all potential inputs, no matter how peculiar

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

Secure coding practices

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

Think about all potential inputs, no matter how peculiar

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    if(i < 0 || i > 9)
        return ‘?’;
    return convert[i];
}
```

Enforce rule compliance at runtime

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

Rule: 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
```

Rule: 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)); //fails
strlcat(buf,” day to you”,sizeof(buf));//fails
```

Rule: 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)); //fails
strlcat(buf,” day to you”,sizeof(buf));//fails
```

Again: know your system’s/language’s semantics

Replacements

- … for string-oriented functions
  - `strcat` ⟹ `strlcat`
  - `strcpy` ⟹ `strlcpy`
  - `strncat` ⟹ `strlcat`
  - `strncpy` ⟹ `strlcpy`
  - `sprintf` ⟹ `snprintf`
  - `vsprintf` ⟹ `vsnprintf`
  - `gets` ⟹ `fgets`

`strncpy`/`strncat` do not NUL-terminate if they run up against the size limit

- Microsoft versions different
  - `strcpy_s`, `strcat_s`, ...

Replacements

- … for string-oriented functions
  - `strcat` ⟹ `strlcat`
  - `strcpy` ⟹ `strlcpy`
  - `strncat` ⟹ `strlcat`
  - `strncpy` ⟹ `strlcpy`
  - `sprintf` ⟹ `snprintf`
  - `vsprintf` ⟹ `vsnprintf`
  - `gets` ⟹ `fgets`

`strncpy/strncat` do not NUL-terminate if they run up against the size limit

- Microsoft versions different
  - `strcpy_s`, `strcat_s`, …

Note: None of these in and of themselves are "insecure." They are just commonly misused.

(Better) **Rule**: Use safe string library

- Libraries designed to ensure strings used safely
  - **Safety first**, despite some performance loss
- Example: Very Secure FTP (**vsftp**) **string library**

http://vsftpd.beasts.org/

```
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);
…
```

- Another example: **C++** **`std::string`** safe string library

Rule: Understand pointer arithmetic

Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

Rule: Understand pointer arithmetic

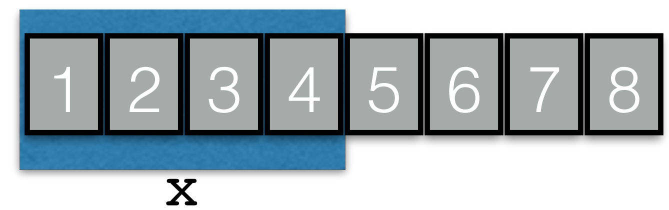
```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

```
(pi + 1) == (pc + 1) ???
```

Rule: Understand pointer arithmetic

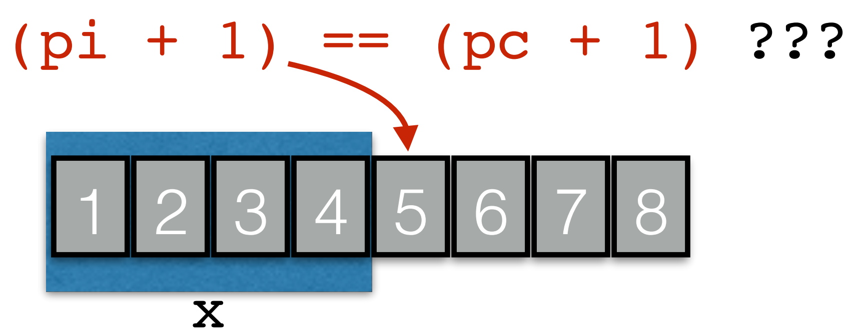
```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

`(pi + 1) == (pc + 1) ???`



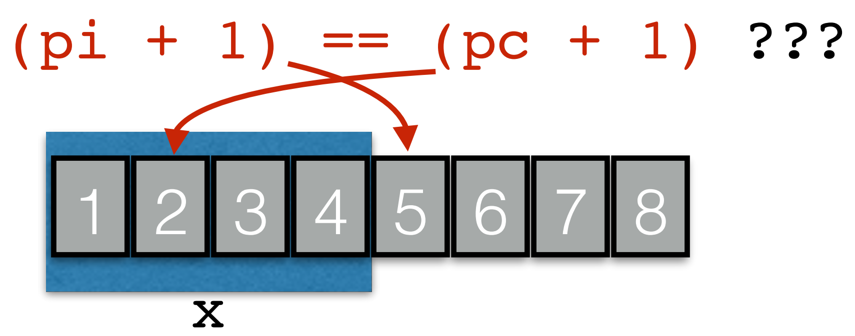
Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```



Rule: Understand pointer arithmetic

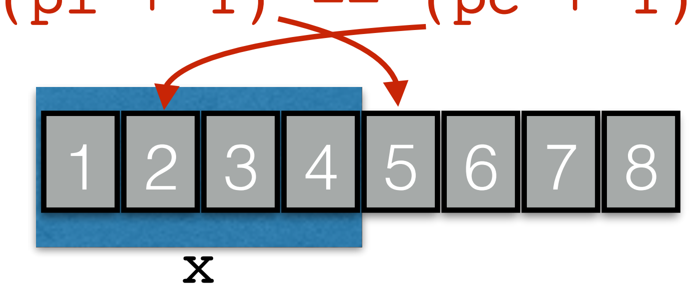
```
int x;
int *pi = &x;
char *pc = (char*) &x;
```



Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

`(pi + 1) == (pc + 1) ???`



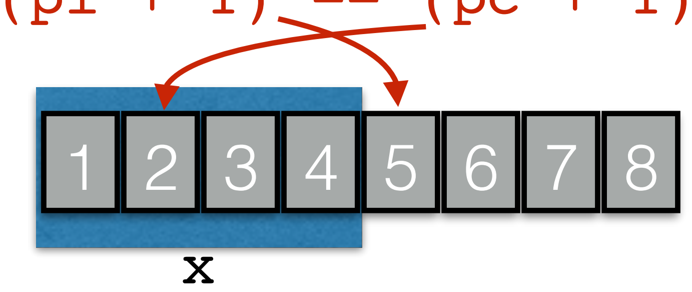
- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

`(pi + 1) == (pc + 1) ???`



- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

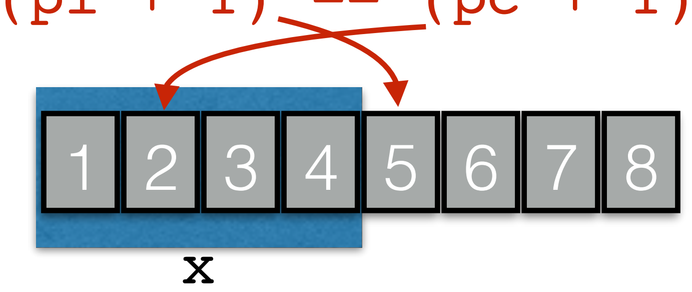
```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

`SIZE * sizeof(int)`

Rule: Understand pointer arithmetic

```
int x;
int *pi = &x;
char *pc = (char*) &x;
```

`(pi + 1) == (pc + 1) ???`



- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

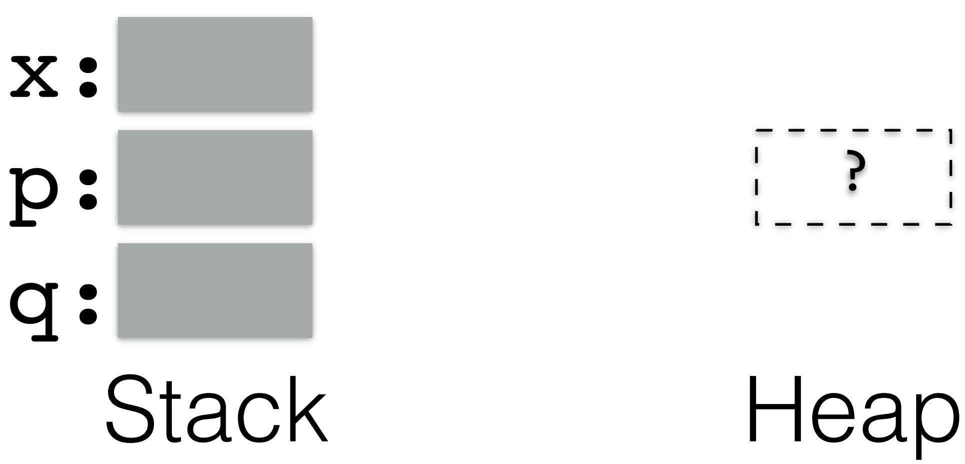
`SIZE * sizeof(int)`

- So, **use the right units**

```
while (!done() && buf_ptr < (buf + SIZE)) {
  *buf_ptr++ = getnext(); // stays in bounds
}
```

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 = 3; //crash (or worse)!
```

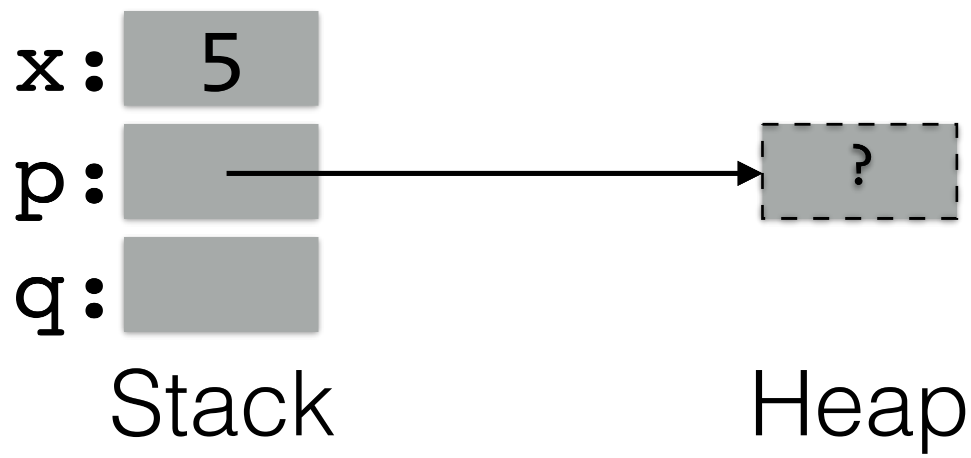


Defend dangling pointers

```
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 = 3; //crash (or worse)!
```

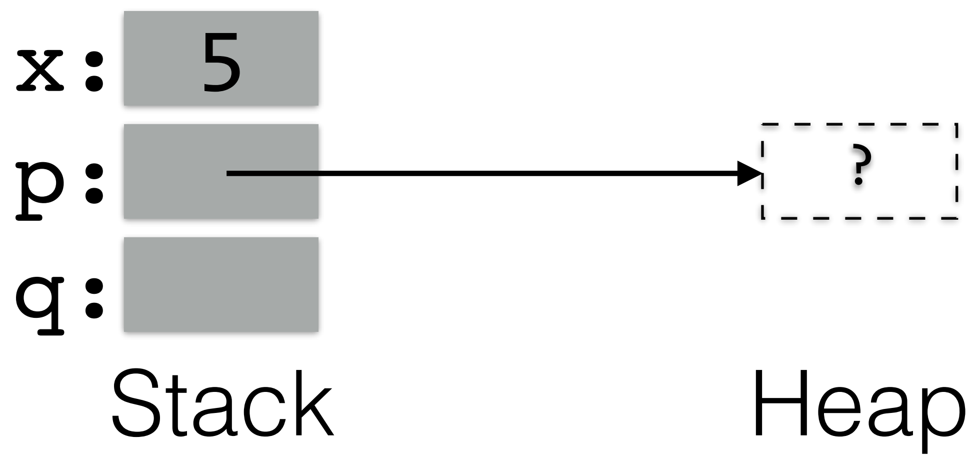
Defend dangling pointers

```
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 = 3; //crash (or worse)!
```



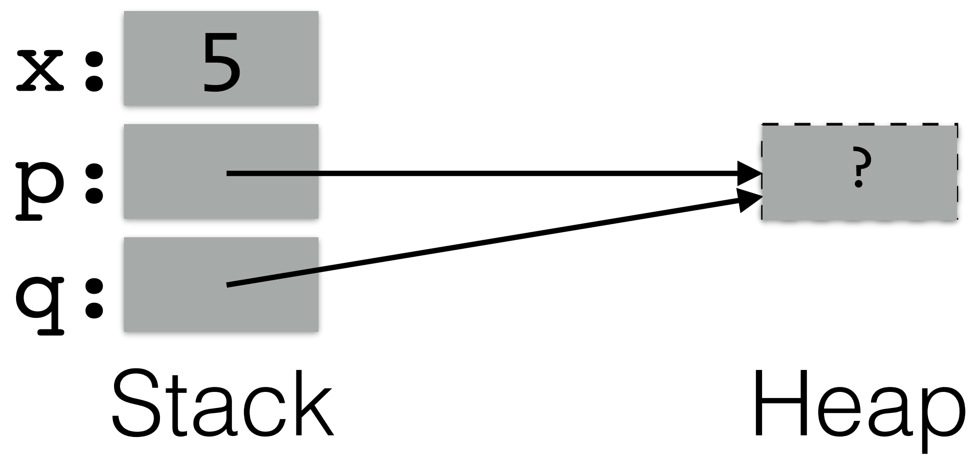
Defend dangling pointers

```
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 = 3; //crash (or worse)!
```



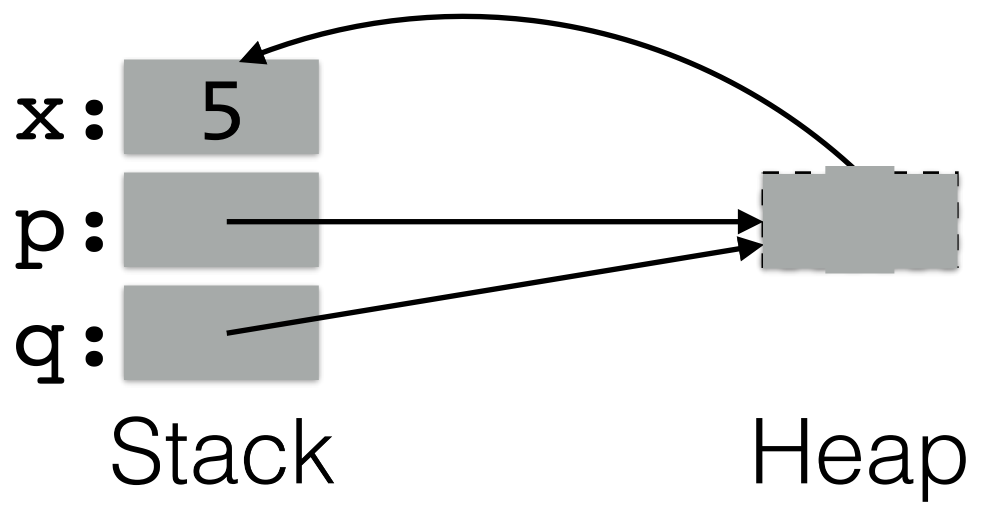
Defend dangling pointers

```
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 = 3; //crash (or worse)!
```



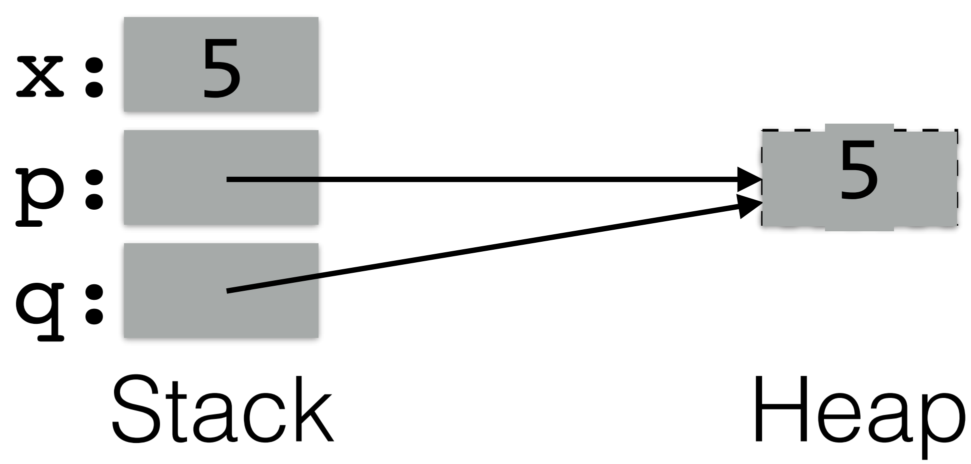
Defend dangling pointers

```
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 = 3; //crash (or worse)!
```



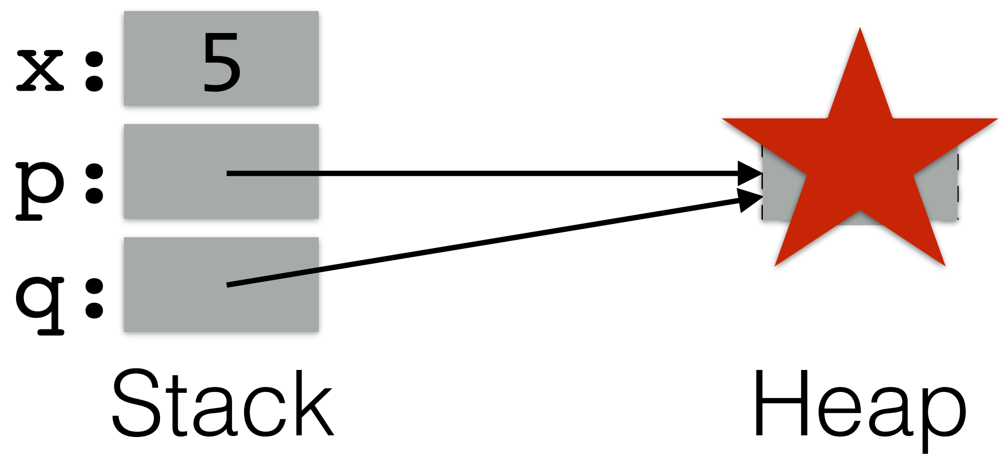
Defend dangling pointers

```
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 = 3; //crash (or worse)!
```



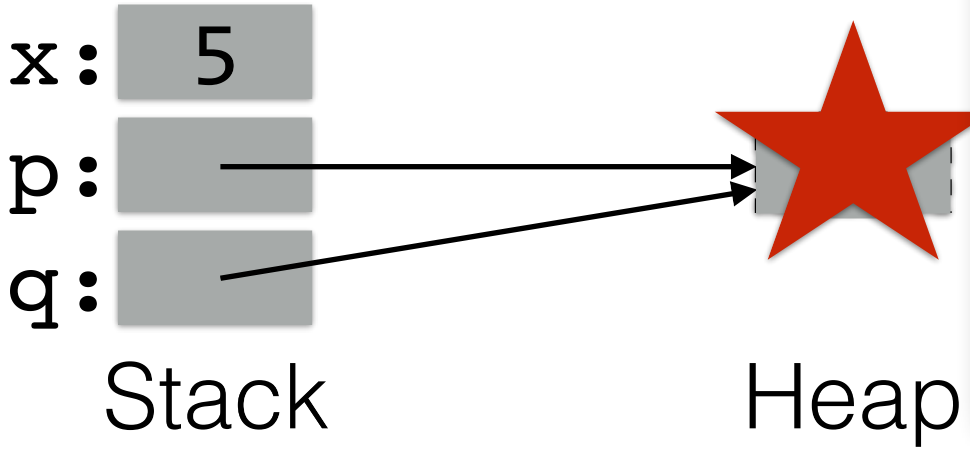
Defend dangling pointers

```
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 = 3; //crash (or worse)!
```



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 = 3; //crash (or worse)!
```



IE's Role in the Google-China War

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

A A Text Size
Print Version
E-Mail Article

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.

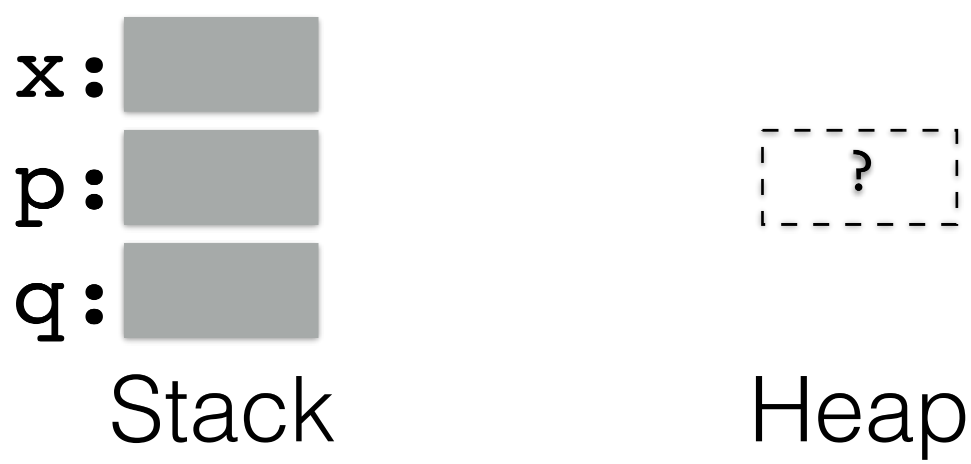
The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

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.

Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```

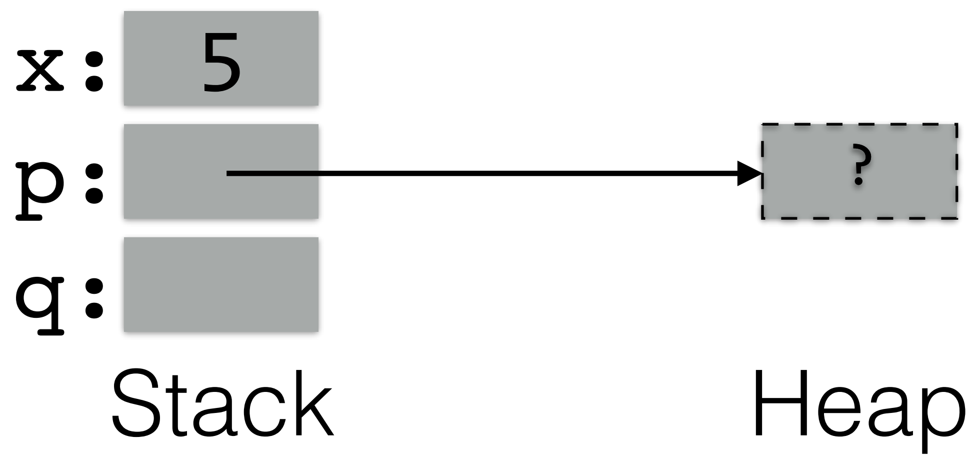


Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```

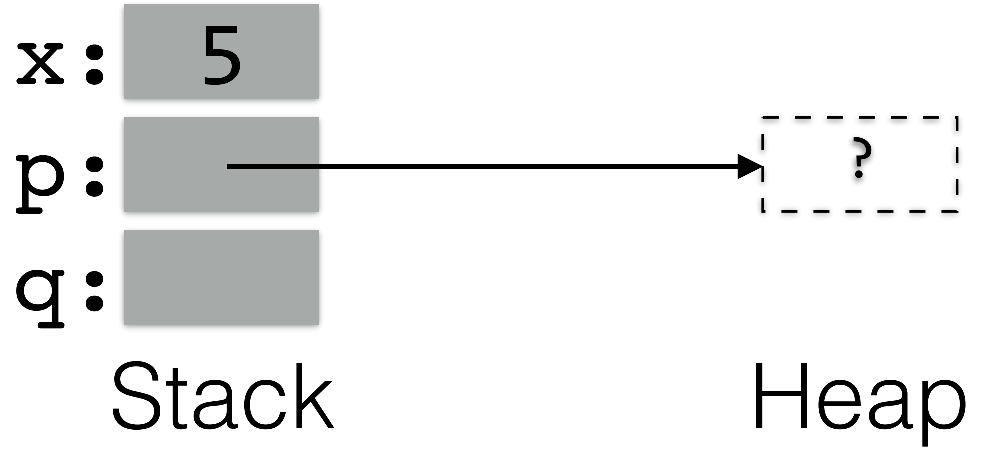
Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```



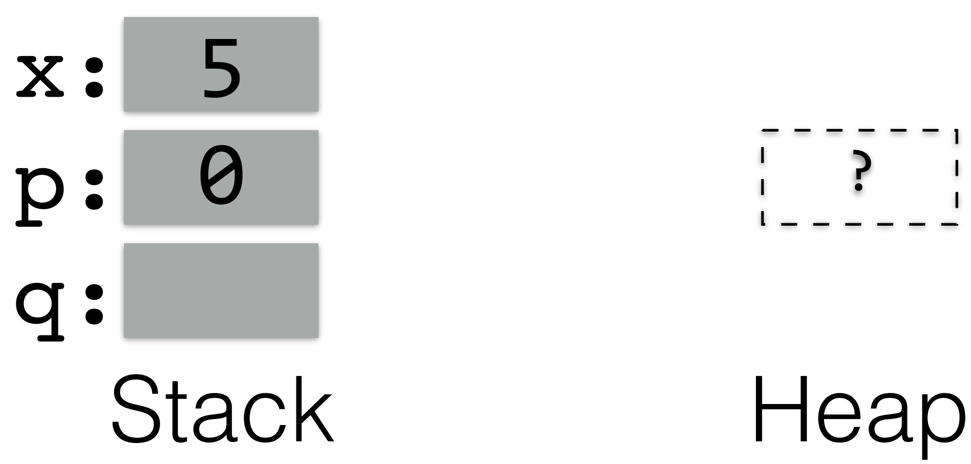
Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```



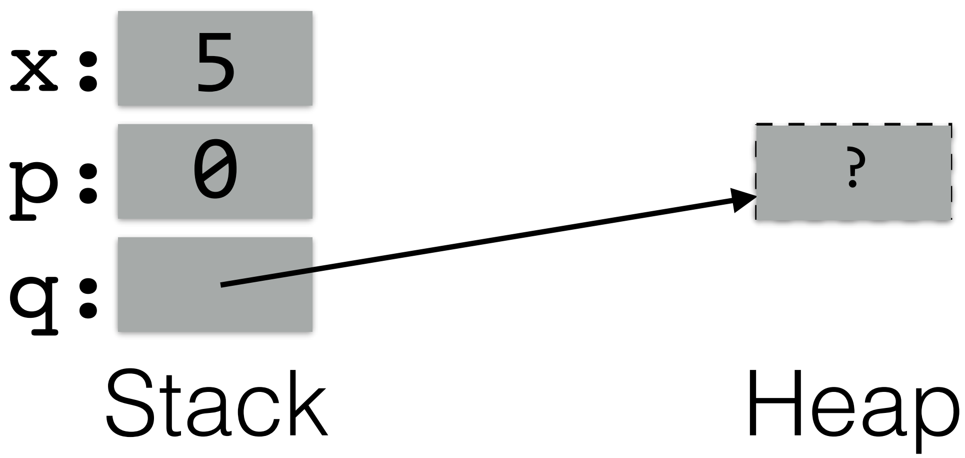
Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```



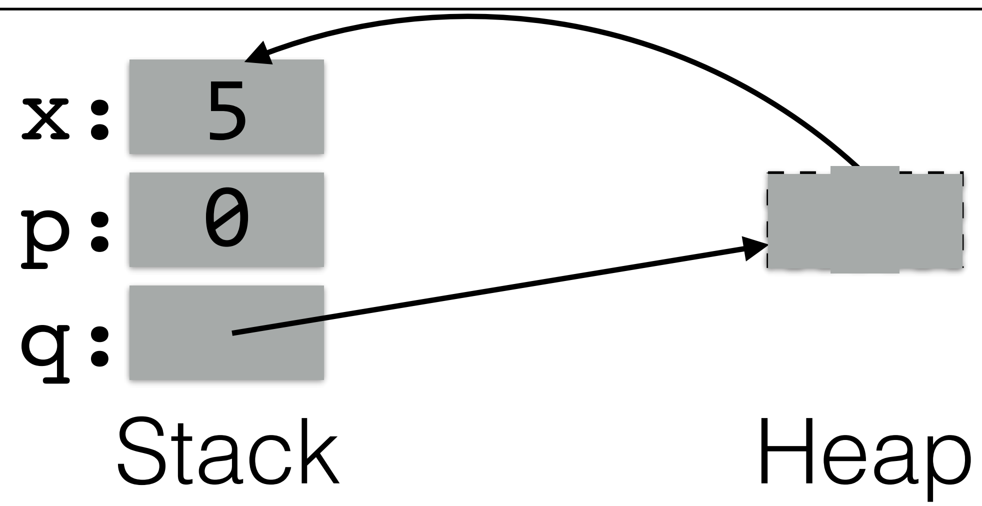
Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```



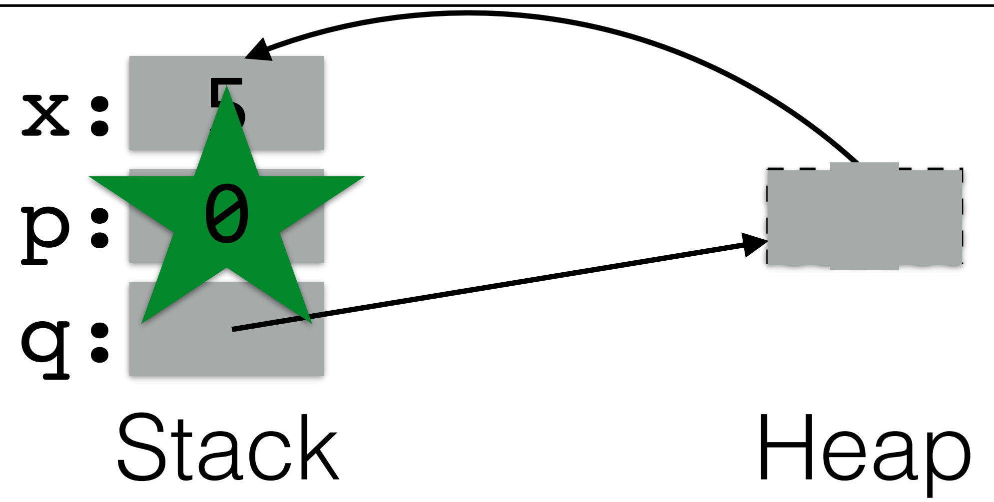
Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```



Rule: Use NULL after free

```
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; //(good) crash
**q = 3;
```



Manage memory properly

```
int foo(int arg1, int arg2) {
  struct foo *pf1, *pf2;
  int retc = -1;

  pf1 = malloc(sizeof(struct foo));
  if (!isok(arg1)) goto DONE;
  …
  pf2 = malloc(sizeof(struct foo));
  if (!isok(arg2)) goto FAIL_ARG2;
  …
  retc = 0;

 FAIL_ARG2:
  free(pf2); //fallthru
 DONE:
  free(pf1);
  return retc;
}
```

- Common approach in C: *goto chains* to avoid duplicated or missed code
  - Like try/finally in languages like Java
- Confirm your logic!…

What’s wrong with this code?

```
{
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .

fail:
    SSLFreeBuffer(&hashCtx);

    return err;
}
```

What's wrong with this code?

```
{
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .

fail:
    SSLFreeBuffer(&hashCtx);

    return err;
}
```

Basically, this is checking a signature (more on this later)

What's wrong with this code?

```
{
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .

fail:
    SSLFreeBuffer(&hashCtx);

    return err;
}
```

Will *always* jump to 'fail'

Basically, this is checking a signature (more on this later)

Rule: 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 implementations:
  - **Windows Fault-Tolerant Heap**
    - http://msdn.microsoft.com/en-us/library/windows/desktop/dd744764(v=vs.85).aspx
  - **DieHard** (on which fault-tolerant heap is based)
    - http://plasma.cs.umass.edu/emery/diehard.html

Rule: Favor safe libraries

- **Libraries** encapsulate **well-thought-out design**. *Take advantage!*

- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard

- **Networking**: Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient

Defensive coding practices

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - 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 (or even exit) than run malicious code

http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Automated testing techniques

- **Code analysis**
  - Static: Many of the bugs we've shown could be easily detected (but we run into the Halting Problem)
  - 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 (e.g., "string int float string")
  - Typically involves *many many* inputs (clusterfuzz.google.com)

Penetration testing

- Fuzz testing is a form of "**penetration testing**" (pen testing)

- Pen testing assesses security by *actively trying to find explotable vulnerabilities*
  - Useful for both attackers and defenders

- Pen testing is useful at many different levels
  - Testing programs
  - Testing applications
  - Testing a network
  - Testing a server…

Kinds of fuzzing

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky

- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space

Kinds of fuzzing

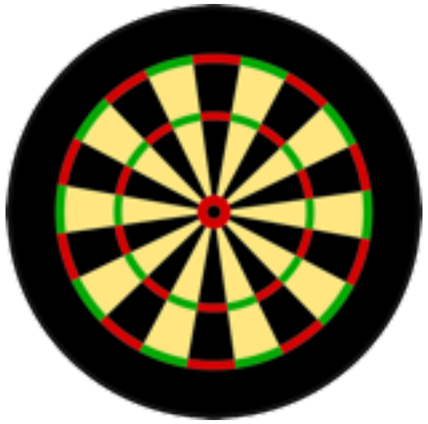
- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky
- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space
- **White box**
  - The tool generates new inputs at least partially informed by the code of the program being fuzzed
  - Often **easy to use**, but **computationally expensive**

Fuzzing inputs

- **Mutation**
    - Take a **legal input and *mutate* it**, using that as input
    - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
        - Mutation might also be forced to adhere to grammar
- **Generational**
    - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**
    - Generate initial input, mutate^{N}, generate new inputs, …
    - Generate mutations according to grammar

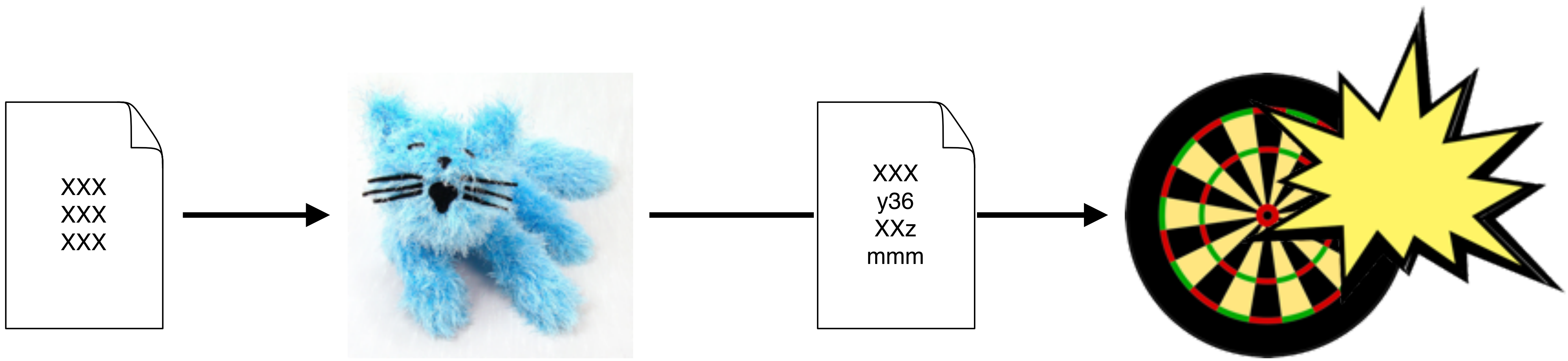
File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- See **what happens**



File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- See **what happens**



Examples: Radamsa and Blab

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
```

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
```

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
```

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

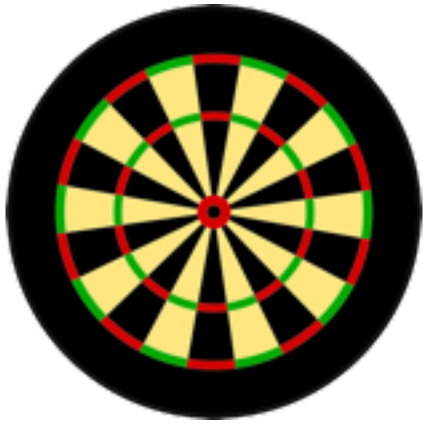
- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
soty wypisi tisyro to patu
```

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

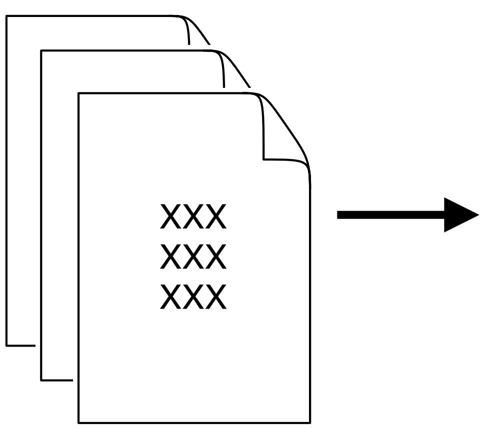
Network-based fuzzing

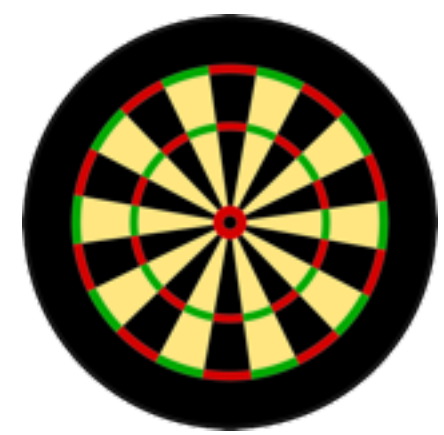
- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)



Network-based fuzzing

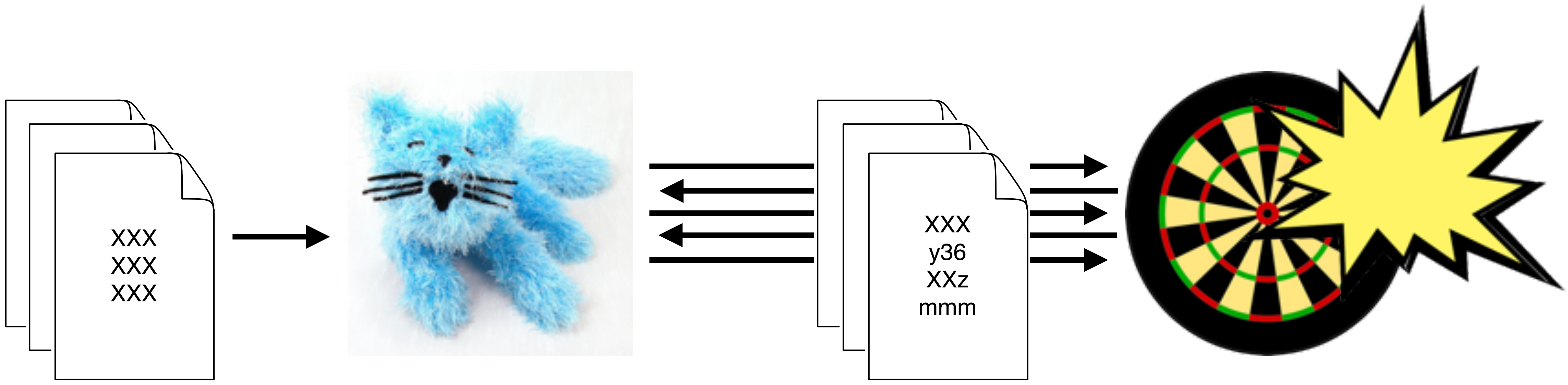
- **Act as 1/2 of a communicating pair**
    - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)





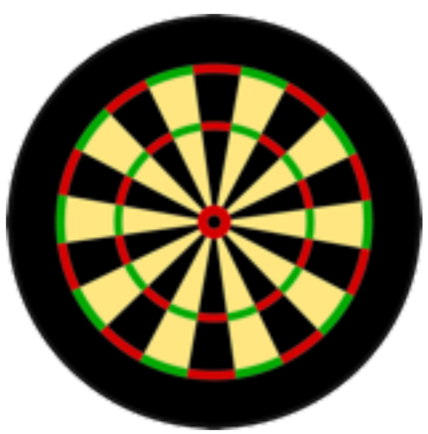
Network-based fuzzing

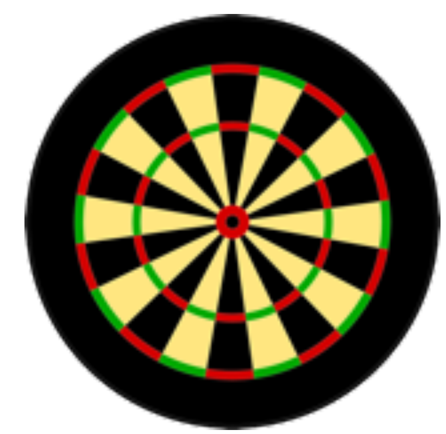
- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)



Network-based fuzzing

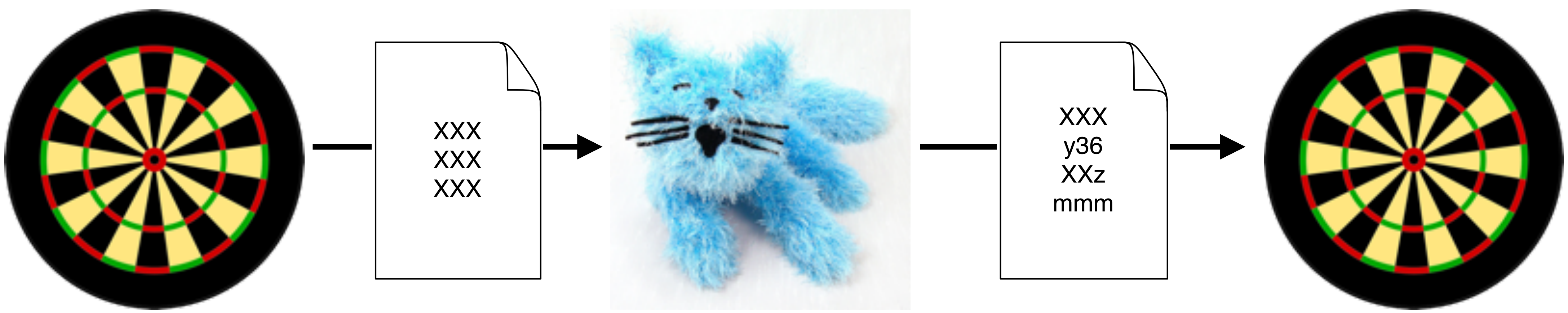
- **Act as a “man in the middle”**
  - mutating inputs exchanged between parties (perhaps informed by a grammar)





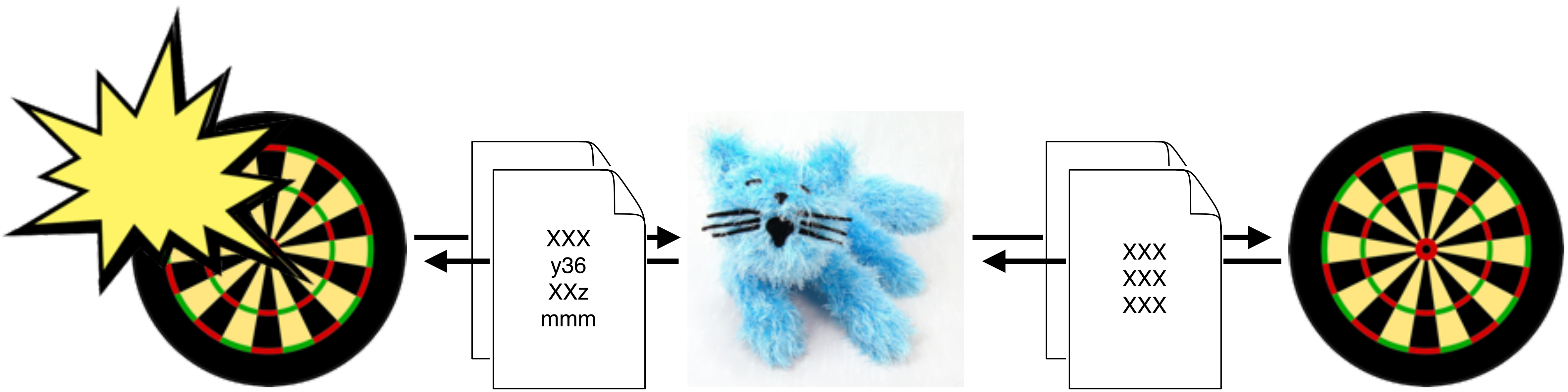
Network-based fuzzing

- **Act as a “man in the middle”**
    - mutating inputs exchanged between parties (perhaps informed by a grammar)



Network-based fuzzing

- **Act as a "man in the middle"**
  - mutating inputs exchanged between parties (perhaps informed by a grammar)



There are many many fuzzers

- American Fuzzy Lop
  - Mutation-based white-box buzzer
- SPIKE
  - A library for creating network-based fuzzers
- Burp Intruder
  - Automates customized attacks against web apps
- And many more… (BFF, Sulley, …)

You fuzz, you crash. Then what?

You fuzz, you crash. Then what?

Try to find the **root cause**

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

Determine if this crash represents an **exploitable vulnerability**

You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

Determine if this crash represents an **exploitable vulnerability**

In particular, is there a buffer overrun?

Finding memory errors

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
    - Instruments accesses to arrays to check for overflows, and use-after-free errors
    - https://code.google.com/p/address-sanitizer/

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/

2. **Fuzz it**

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/

2. **Fuzz it**

3. Did the program **crash with an ASAN-signaled error**? Then worry about exploitability

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/

2. **Fuzz it**

3. Did the program **crash with an ASAN-signaled error**? Then worry about exploitability

- Similarly, you can *compile with other sorts of error checkers* for the purposes of testing
  - E.g., `valgrind` memcheck   http://valgrind.org/

Writing and testing for secure code

- Know the systems and libraries you use
  - printf("%n"); !!
- Assume the worst; code and design defensively
  - Think defensive driving
- Use ***pen testing*** tools to help automate
  - Assume that attackers will; seek to have at least as much information as they!

Up next

Continuing with
Software Security

Getting sick with
Malware

Required reading:

"StackGuard: Simple Stack Smash Protection for GCC"

Optional reading:

"Basic Integer Overflows"

"Exploiting Format String Vulnerabilities"

http://nob.cs.ucdavis.edu/bishop/secprog/robust.html