Statically Detecting Likely Buffer Overflow Vulnerabilities

David Larochelle and David Evans
USENIX'01
David Larochelle and David Evans
IEEE Software Jan/Feb 2002

Presented by Adam Polyak
30.03.2014

Outline of talk

- Introduction
- Suggested Solution: Splint
- Evaluation
- Related work
- Conclusions

Introduction

```
int B=0;
char A[8]={};
strcpy(A, "excessive");
```

- Before command:

Var name	A								B	
Value	[empty]								0	
Hex Value	00	00	00	00	00	00	00	00	00	00

- After command:

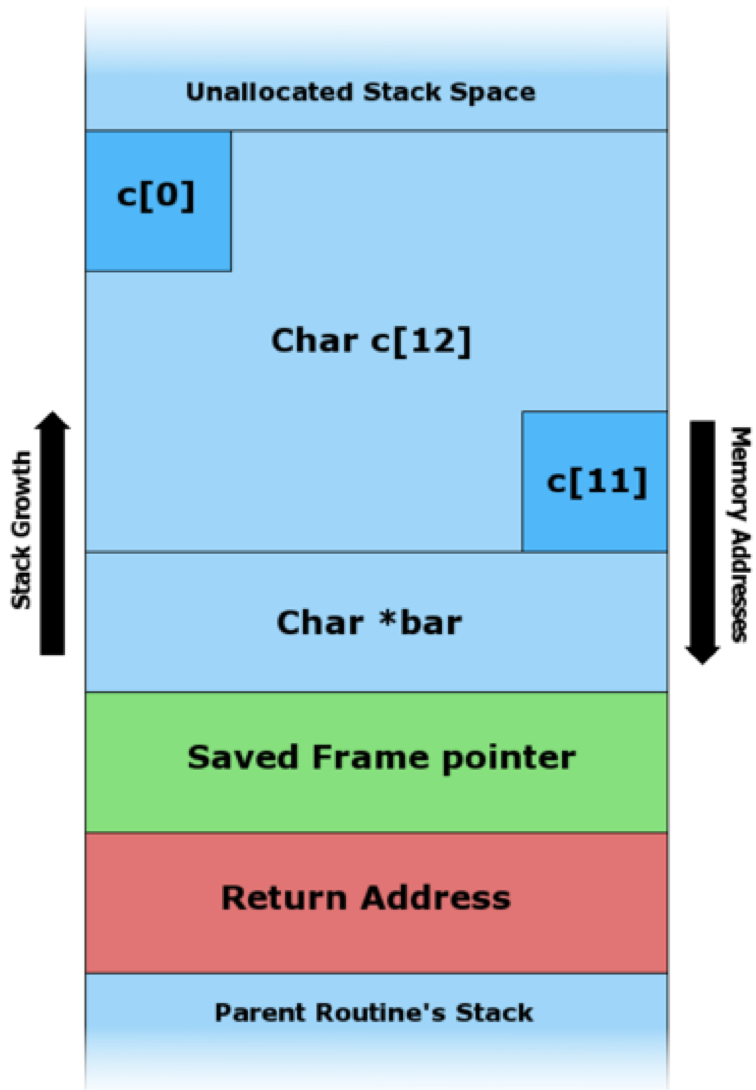
Var name	A								B	
Value	‘e’	‘x’	‘c’	‘e’	‘s’	‘s’	‘i’	‘v’	25856	
Hex Value	65	78	63	65	73	73	69	76	65	00

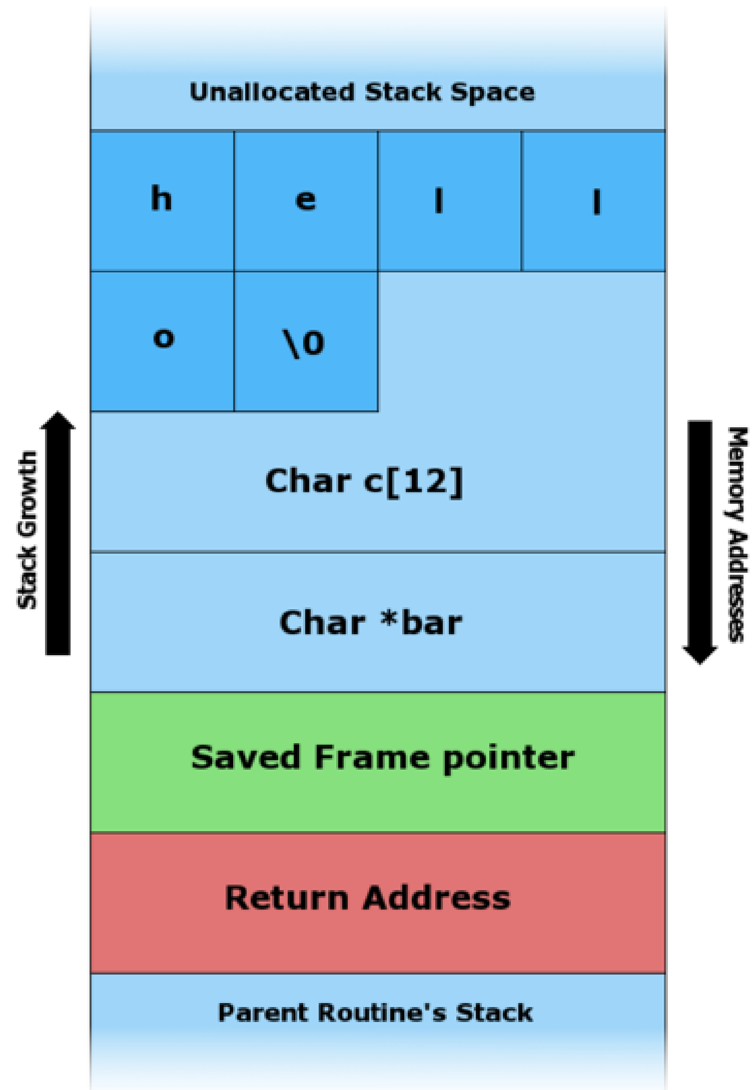
It can get worse

- Code execution

```
void foo (char *bar){
    char  c[12];
    strcpy(c, bar);
}
int main (int argc, char **argv) {
    foo(argv[1]);
}
```

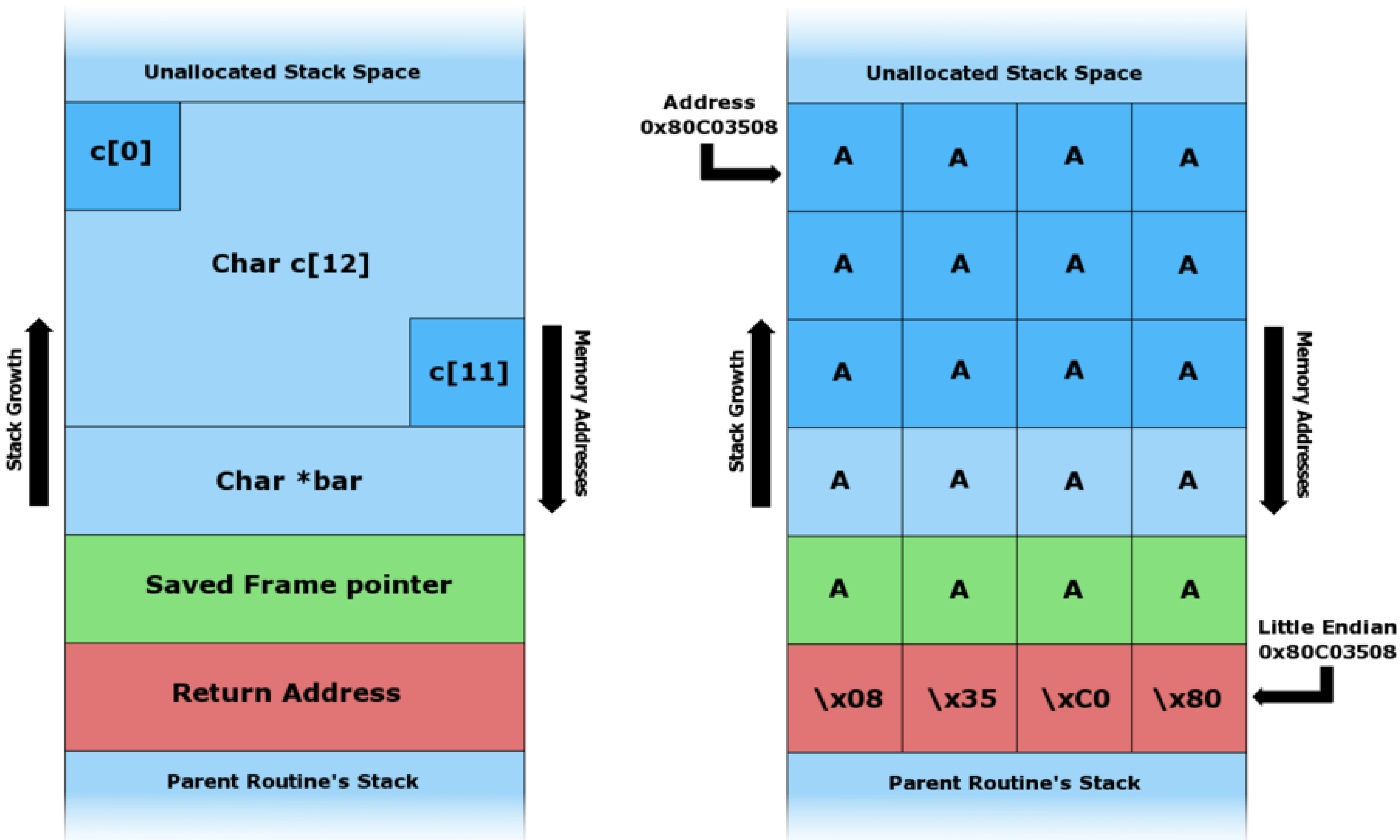
It can get worse





http://en.wikipedia.org/wiki/Stack_buffer_overflow

It can get worse



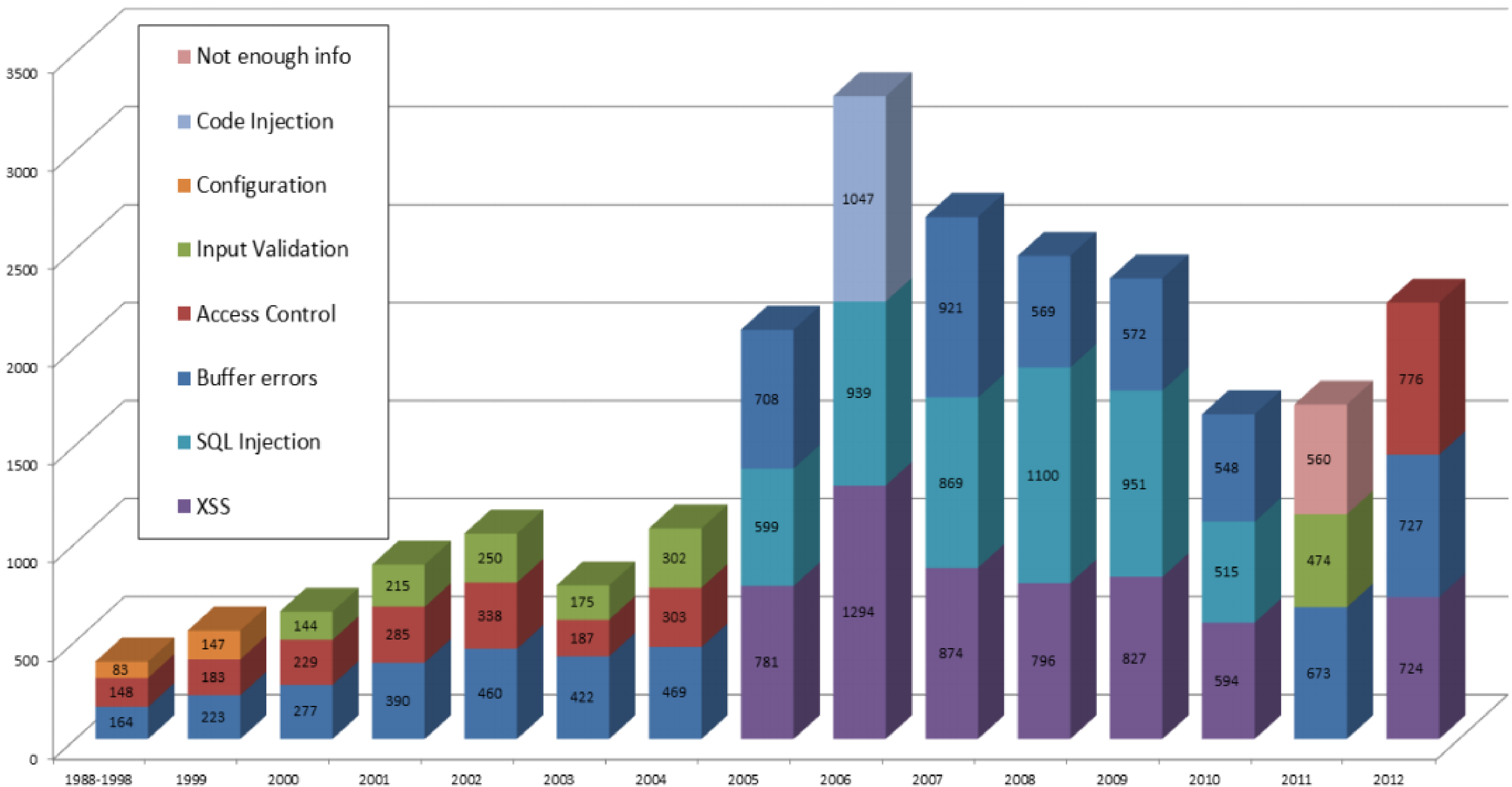
http://en.wikipedia.org/wiki/Stack_buffer_overflow

It can get worse

- Why limit ourselves to stack? Heap buffer overflow
  - Heap is a linked list
  - What if we corrupt one of the links?

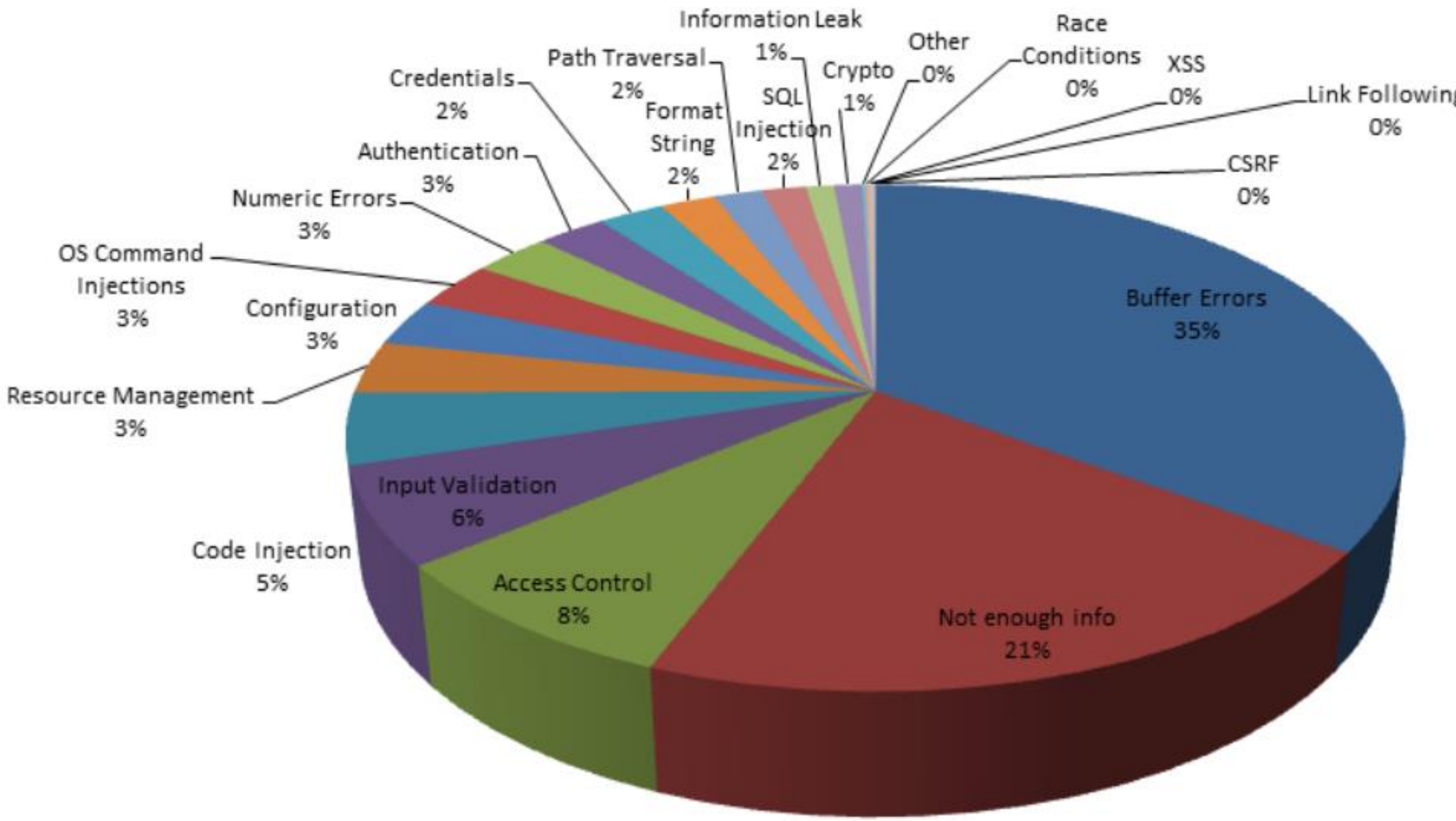
```
#define unlink( y, BK, FD ) {
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```

Why Is It Important?



RSAConference2013 – 25 Years of Vulnerabilities

Critical Vulnerabilities By Type



RSAConference2013 – 25 Years of Vulnerabilities

Causes

- Programs written in C
  - Prefer space and performance over security
  - Unsafe language – direct access to memory
- Lack of awareness about security
  - Code is written to work
- Legacy code
  - Knowledge about code isn’t preserved
- Inadequate development process

Defense: Limiting Damage

- Mostly runtime methods:
  – Compiler modifications (stack cookies) - StackGuard
  – Safe libraries - Baratloo, Singh and Tsai
  – Modify program binaries - (assert) SFI
- Pros:
  – Minimal extra work is required from developers
- Cons:
  – Increase performance/memory overhead
  – Simply replace the flaw with a DOS vulnerability

Defense: Eliminating Flaws

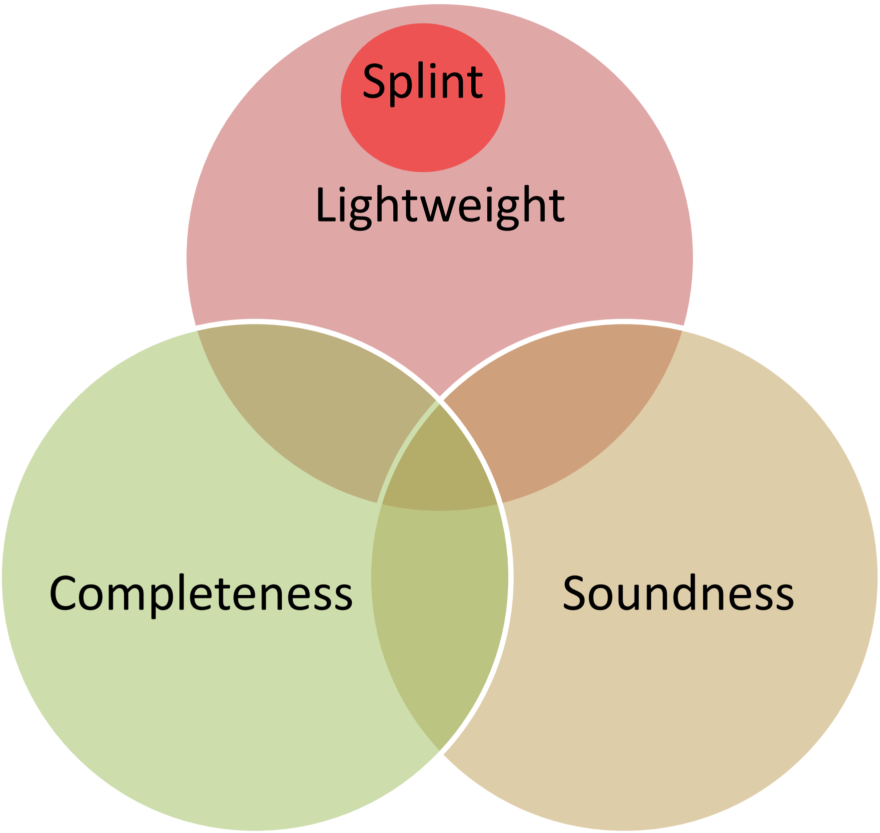
- Human code review
  - Better than automatic tools
  - Can overlook problems
- Testing
  - Mostly ineffective (security wise)
  - Checks expected program behavior
- Fuzzing – doesn’t check all possibilities
- Static analysis
  - Allows to codify human knowledge

Defense: Static Analysis

- Pros:
  - Analyze source code directly, lets us make claims about all possible program paths
  - It still possible to generate useful information
- Cons:
  - Detecting buffer overflows is an undecidable problem
  - Wide range of methods:
    - compilers (low-effort, simple analysis)
    - Full program verifiers (expensive yet effective)

Solution: Splint – lightweight static analysis

- "Useful results for real programs with reasonable effort"



Solution: Splint – main ideas

- Static Analysis:
  - Use semantic comments to enable checking of interprocedural properties
  - Use loop heuristics

- Lightweight:
  - good performance and scalability
  - sacrifices soundness and completeness

Annotations

- Splint is based upon LCLint
  - Annotation assisted static checking tool
- Describe programmer assumptions and intents
- Added to source code and libraries
- Associate with:

Function parameters	Local variables
Function return values	Structure fields
Global variables	Type definitions

Example: @null@

```c
1 typedef /*@null@*/ char *mstring;
2 static mstring mstring_createNew (int x) ;
3 mstring mstring_space1 (void) {
4   mstring m = mstring_createNew (1);
5   /* error, since m could be NULL */
6   *m = ' '; *(m + 1) = '\0';
7   return m;
8 }
```

mstringnn.c: (in function mstring_space1)
mstringnn.c:6,4: Dereference of possibly null pointer m: *m

Example: @notnull@

```
static /*@notnull@*/ mstring
mstring_createNewNN (int x) ;
mstring mstring_space2 (void) {
   mstring m = mstring_createNewNN (1);
   /* no error, because of notnull annotation */
   *m = ' ';
   *(m + 1) = '\0';
   return m;
}
```

Annotations: Buffer Overflow

- In LCLint – references are limited to a small number of possible states.
- Splint extends LCLint to support a more general annotation
- Functions pre(`requires`) and post (`ensures`) conditions
- Describe assumptions about buffers:

`minSet`	`maxSet`
`minRead`	`maxRead`

Example: Buffer Annotations

- The declaration:

```
char buf[MAXSIZE];
```

Generates the **constraints**:

```
maxSet(buf) = MAXSIZE - 1
minSet(buf) = 0
```

- Functions **conditions**:

```
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead(s2)
/\ result == s1@*/;
```

Annotations: Buffer Overflow Cont.

- **Constraints** are used to validate **conditions**
- Conditions are found 3 ways:
  - By Splint:

`buf[i] = 'a'`	Precondition : `maxSet(buf) >= i`
`char a = buf[i]`	Precondition: `maxRead(buf) >= i`
`buf[i] = 'a'`	Postcondition: `maxRead(buf) >= i`

  - Library functions are annotated
  - User generated

Example: Buffer Overflow Conditions

```
1    void updateEnv(char * str)
2    {
3        char * tmp;
4        tmp = getenv(MYENV);
5        if (tmp != NULL)
6            strcpy(str, tmp);
7    }
```

<u>Unable</u> to resolve constraint:

requires maxSet(str @ bounds.c:6) >=
maxRead(getenv("MYENV")@bounds.c:4)

<u>needed</u> to satisfy precondition:

requires maxSet(str @ bounds.c:6) >= maxRead(tmp @ bounds.c:6)

<u>derived</u> from strcpy precondition:

requires maxSet(<parameter 1>) >=maxRead(<parameter 2>)

Analysis: Settings

- Static analysis is limited
  - Depends on several undecidable problems
- Unsound -> False positives
- Incomplete -> False negatives
- Scalability over precision
- Highly configurable for users

Analysis: Settings cont.

- Analysis is mostly intraprocedural
  - i.e. at function level
  - Achieve interprocedural (dataflow between functions) using annotations
- Flow-sensitive (order of statements matters) with compromises
  - Handle loops using heuristics

Analysis: Algorithm

- Programs are analyzed at the function level
- Generate constraints for each C statement
  - By conjoining the constraints of sub expressions
  - Simplify constraints: maxSet(ptr+ i) = maxSet(ptr) - i
- Constraints are resolved at call site
  - Done at statement level
  - Use function preconditions and postconditions of earlier statements

Analysis: Algorithm Cont.

- All variables used in constraints have an associated location

```
1    t++;
2    *t = x;
3    t++;
```

- Leads to the constraints:

  requires maxSet(t @ 1:1) >= 1,

  ensures maxRead(t @ 3:4) >= -1 and

  ensures (t @ 3:4) = (t @ 1:1) + 2

Axiomatic semantics

- Mathematical logic for proving the correctness of computer programs

$$\{\ \underbrace{P}_{\text{precondition}}\ \}\ \underbrace{C}_{\substack{\text{statement}\\ \text{a.k.a command}}}\ \{\ \underbrace{Q}_{\text{postcondition}}\ \}$$

- P & Q are state predicates, C a command
- ***If*** P holds and ***if*** C terminates, ***then*** Q will hold

Slide credit [:http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html](http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html)

Analysis: Control Flow - If

- Condition and loops can make unsafe operations, safe
- Analyze an `if` based on the predicate, for example:

```
if (sizeof (s1) > strlen (s2))
    strcpy(s1, s2);
```

  – If the condition holds, the operation is safe

Analysis: Control Flow - Loops

- Statically analyzing loops is hard. Simplify:
  - Analyze loops according to common idioms
  - First and last iteration matter
- First iteration:
  - Treated as `If` statement
- Last iteration:
  - Determine number of runs base on loop heuristics:
    - **for (**index**=**0**;**expr**;**index**++)** body
    - **for (**init**;***buf**;**buf**++)**

Evaluation

- Analyzed two popular programs
  - wu-ftpd
  - BIND
- Checked detection of known and unknown buffer overflows

Evaluation: wu-ftpd

- Version wu-ftp-2.5.0 with known security vulnerabilities
- Execution:

Code size(lines)	Time(minutes)
17,000	1

- Source code wasn't modified – resulted 243 warnings(related to buffer overflow)
- Detected both known and unknown buffer overflows

wu-ftpd: Unknown Bug

```
char ls_short[1024];
…
extern struct aclmember *
getaclentry(char *keyword,
            struct aclmember **next);
…
int main(int argc, char **argv,
         char **envp)
{
…
entry = (struct aclmember *) NULL;
if (getaclentry("ls_short", &entry)
    && entry->arg[0]
    && (int)strlen(entry->arg[0]) > 0)
{
strcpy(ls_short,entry->arg[0]);
…
```

wu-ftpd: Unknown Bug Cont.

- Will generate the following warning:

  Possible out-of-bounds store.

  <u>Unable</u> to resolve constraint:

      maxRead ((entry->arg[0] @ ftpd.c:1112:23)) <= (1023)

  <u>needed</u> to satisfy precondition:

      requires maxSet ((ls_short @ ftpd.c:1112:14))

          >= maxRead ((entry->arg[0] @ ftpd.c:1112:23))

  <u>derived from</u> strcpy precondition:

      requires maxSet (<param 1>) >= maxRead (<param 2>)

wu-ftpd: Known Bug

```
char mapped_path [200];
…
void do_elem(char *dir) {
…
if (!(mapped_path[0] == '/'
        && mapped_path[1] == '\0'))
            strcat (mapped_path, "/");
            strcat (mapped_path, dir);
}
```

- `dir` is entered by a remote user
- Reported buffer overflow:
  - http://www.cert.org/historical/advisories/CA-1999-13.cfm

wu-ftpd: False positive

```
i = passive_port_max - passive_port_min + 1;
port_array = calloc (i, sizeof (int));

for (i = 3; … && (i > 0); i--) {
  for (j = passive_port_max
             - passive_port_min + 1;
         … && (j > 0); j--) {
             k = (int) ((1.0*j*rand()) /
                        (RAND_MAX + 1.0));
             pasv_port_array [j-1]
                = port_array [k];
```

- Unable to determine that `1 < k < j`
- Can be suppressed by the user

wu-ftpd: Summary

- Unmodified code:
  - 243 warnings
- After adding 22 annotations
  - 143 warnings
  - Of these, 88 unresolved constraints involving maxSet
  - The rest, can be suppressed by the user
- 225 calls to unsafe functions(strcat, strcpy, …)
  - Only 18 reported by Splint
  - 92% of the calls are safe by Splint

Evaluation: BIND

- Berkley Internet Name Domain – reference for DNS implementations
- Version 8.2.2p7
- Execution:

Code size(lines)	Time(minutes)
47,000	3.5

- Check limited to a subset of code(~3,000 lines)
  - Because of time of human analysis

BIND: Library Annotations

- Extensive use of internal libraries instead of C library functions
  - Requires annotating large code base
  - Iteratively run Splint and annotate
- To reduce human analysis required
  - Only interface library functions were annotated -> based on code comments and parameters names
- For example:
  - int foo**(**char***** p**,** int size**)**
  - Resulted: MaxSet(p) >= (psize - 1)

BIND: Library Annotations

- Extensive use of internal libraries instead of C library functions
  - Requires annotating large code base
  - Iteratively run Splint and annotate
- To reduce human analysis required
  - Only interface library functions were annotated -> based on code comments and parameters names
- For example:
  - int foo**(**char***** p**,** int size**)**
  - Resulted: MaxSet(p) >= (psize - 1)

BIND: req_query

- Code called in response for querying the domain name server version
- Version string read from configuration file and appended to a buffer
  - OK if used with default
- However, sensitive to
  - Code modification
  - Configuration changes

BIND Cont.

- BIND uses extensive run time bounds checking
  - This doesn’t guarantee safety
  - Buffer overflow was detected, because buffer sizes were calculated incorrectly
- ns_sign – the functions receives a buffer and its size
  - Splint detected that the size might be incorrect
  - Occurs, If the message contains a signature but the key isn’t found
  - Bug was introduced after feature was added

Related work – Lexical analysis

- Idea:
  - Put simply , use **grep** to find unsafe code
- Tool:
  - ITS4 - [VBKM00]
- Pros:
  - simple and fast
- Cons:
  - Very limited, deserts semantics and syntax

Related work – Proof-carrying code

- Idea [NL 96, Necula97] :
  - Executable is distributed with a proof and verifier
  - Ensures the executable has certain properties
- Pros:
  - Sound solution
- Cons:
  - At time of paper, wasn’t feasible automatically

Related work – Integer range analysis

- Idea [Wagner et al]:
  - Treat strings as integer range
- Cons:
  - Non-character buffers are abandoned
  - Insensitive analysis – ignore loops and conditions

Related work – Source transformation

- Idea [Dor, Rodeh and Sagiv]:
  - Instrument the code
  - Assert string operations
  - Then use integer analysis
- Pros:
  - Can handle complex properties – pointer overlapping
- Cons:
  - Doesn’t scale

Conclusions

- Splint isn't perfect but improves security with a reasonable amount of effort
- Splint is lightweight
  - Scalable
  - Efficient
  - Simple
- Hard to introduce to a large code base
  - However, done incrementally can be just right

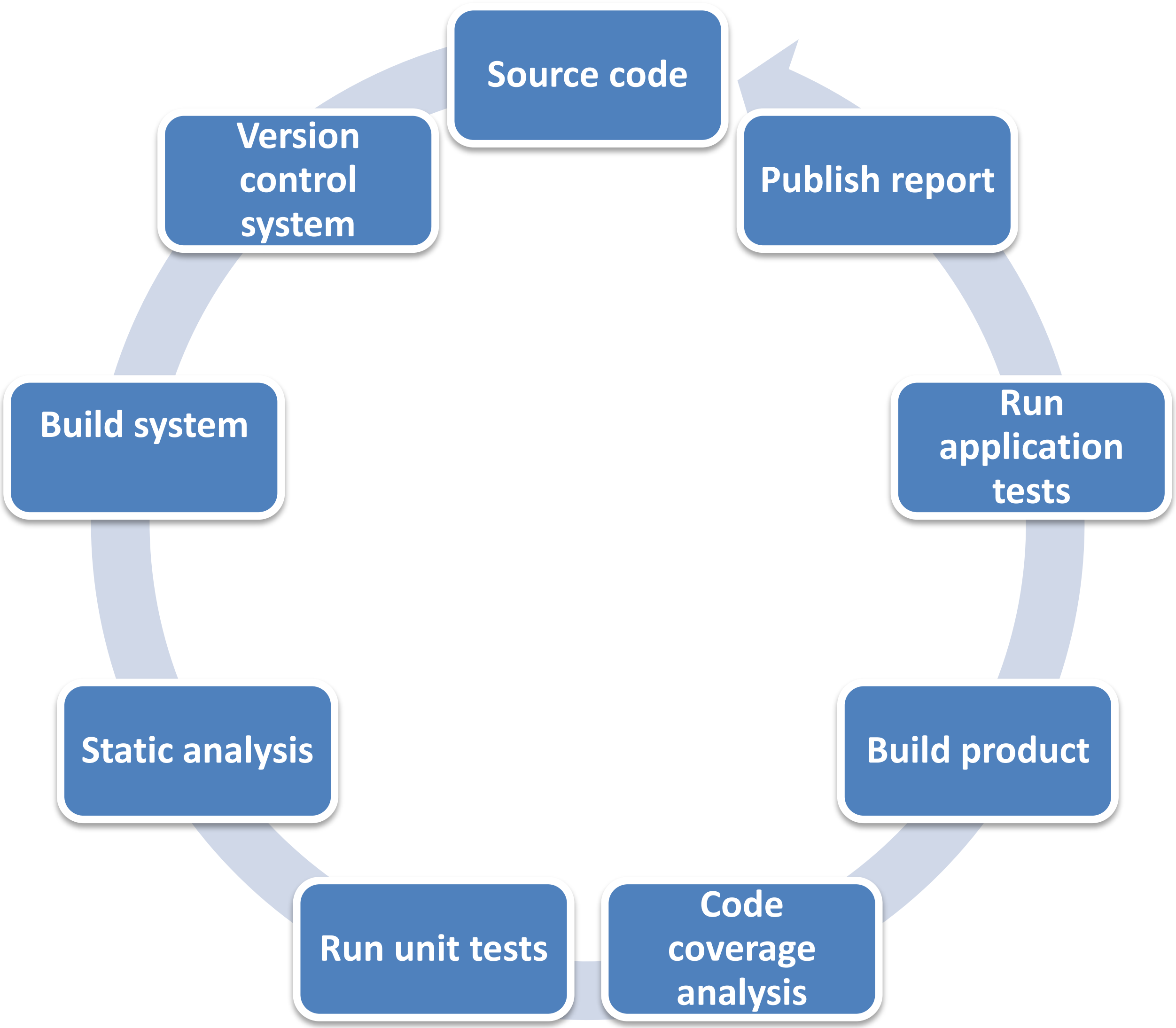
Reflections

- Practical approach!
  - 80/20 principle in action
- Human effort vs security gained
- Improves code base readability and maintainability
- Can be leveraged as part of Continuous Integration process – Code Quality

Continuous integration

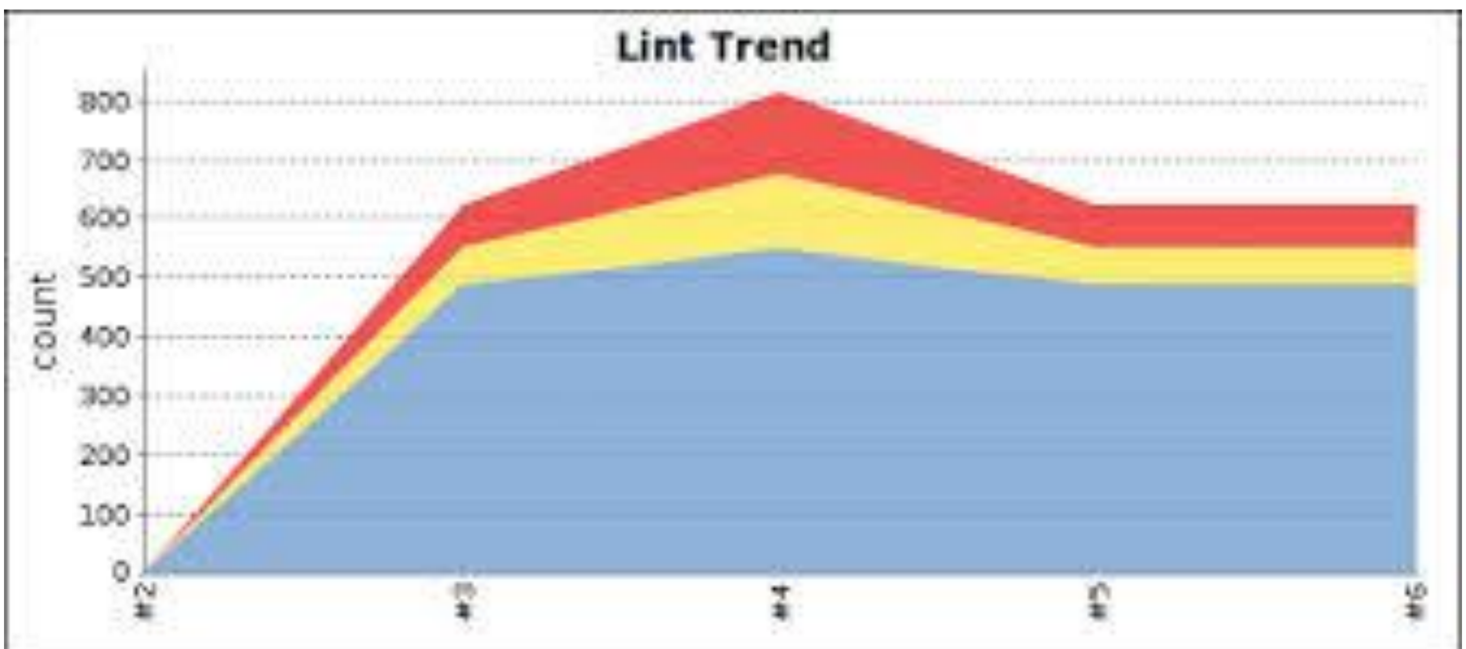
- "The practice of frequently integrating one's new or changed code with the existing code repository"
- Some of its advantages:
  - Immediate unit testing of all changes
  - Early warning of broken/incompatible code
  - Frequent code check-in pushes developers to create modular, less complex code

Continuous integration



Continuous integration

- Static analysis on source code:
  - Ensures coding standards
  - Assists in avoiding common bugs



Questions?

Discussion

- Would you use Splint\LCLint in your projects?
  - Security wise?
  - Code quality wise?
- What about dynamic languages?
  - Not necessarily security