Finding and Preventing Buffer Overflows

An overview of scientific approaches

Martin Johns
Me, myself and I

- Martin Johns
- johns at informatik.uni-hamburg.de
- Security researcher at the University of Hamburg
- Member of the secologic project
  - Research project carried out by SAP, Commerzbank, Eurosec and the University of Hamburg
  - Goal: Improving software security
  - Visit us at http://www.secologic.org
Overview (I)

- Only tools that are applicable by the programmer are presented
  - There are also counter measure that can be applied by the administrator of the application
- (mostly) tools with origin in academia
- Commercial tools exist but are rarely explicit about their internal algorithms
Most approaches concentrate on the C language

- C is widely used (especially for system programming)
- A lot of security problems are caused by C programs
- C is “easy” to check
  - The control flow of a program is (mostly) determined on compile time
  - C programs are often vulnerable on a syntactic level
- C is “hard” to check
  - Pointer arithmetic and type casting
  - Heavy use of preprocessor
  - More than one C

Depending on the tool different kinds of vulnerabilities are detected / prevented

- Buffer Overflows
- Heap Corruption
- Format String Exploits
A classification

Static: check is done before / during compilation
Dynamic: check is done on runtime
A classification (teaser)

Approaches

Static
  - Syntax orientated
  - Compiler related
  - Theory based

Dynamic
  - Detection
  - Prevention

Finite automata
  - Program verification
A classification

Approaches

- static
  - syntax orientated
  - compiler related
  - theory based
  - finite automatons

- dynamic
  - detection
  - prevention
  - program verification
A lot of security problems of C programs are caused by “unsafe” library functions

Example:
```
strcpy(dst,sr)  
```

These functions are comparably easy to spot

Internals:

- Syntactic tools examine on a per statement basis
- Usually these kind of tools operate on an internal representation of the source code
  - E.g. token stream, AST, etc.
- This helps to eliminate obvious sources of false positives
  - Comments
  - Strings
Static tools: syntactic analysis (II)

Some tools:

- **Flawfinder (2001)**
  - Written by David Wheeler (Author of “Secure Programming for Linux and Unix”)
  - Displays the code context of the found vulnerable constructs

- **ITS4 (2001)**
  - Assigns severity levels to warning
  - Also checks for some TOCTOU-problems

- **RATS (2001)**
  - Differentiates between heap- and stack allocated buffers
  - Dictionaries for C(++) , Python, Perl and PHP
Limitations of syntactic analysis:

- Only a limited context is taken into account
  - (sometimes) type qualifier (e.g. `strcpy` with a `const` source buffer is not exploitable)
  - (sometimes) preliminary checks

- Complex contexts are ignored
  - Intra-/Interprocedural dependencies
  - control flow
  - data flow

- Consequences
  - Syntactic analysis is prone to false positives (e.g. every `strcpy()` gets reported)
  - Syntactic analysis is unable to find problems of higher semantic level (e.g. “double free”, access violations, etc.)
A classification

- Approaches
  - static
    - syntax orientated
    - compiler related
  - dynamic
    - theory based
    - detection
    - prevention
      - finite automatons
      - program verification
Static Tools: compiler related approaches

- General observation: static analysis tools and compilers share common techniques
- Compiler actions:
  - Parsing source code to abstract representation (token stream, AST, etc.)
  - Generating control- and data-flow graphs (for optimization)
  - Enforcement/check of constraints (e.g. type checks)
- The more advanced the compiler is, the better its “understanding” of the source code’s semantics → interesting aspect for security related analysis
Compiler related approaches: BOON

Buffer Overrun detection (2000)

- Introduces a theoretical “C String” abstract data type consisting of char-buffer & string-library functions
- Examines code for potential C String overflows
  → Only char-buffers are considered
  → Only Buffer Overruns caused by library functions are detected
- Ignores control flow
- The state of every C-String s is represented as two integer ranges
  alloc(s) and len(s) = [min, max]
- The safety property to be verified is
  max(len(s)) <= min(alloc(s))
- Integer range algebra:
  a ⊆ b → b = [min(min(a),min(b)), max(max(a), max(b))]
  example: a = [2,5]; [4,7] ⊆ a → a = [2,7]
- For each statement an integer range constraint is constructed:
  s = malloc(6*sizeof(char)); → [6,6] ⊆ alloc(s),
  fgets(s,n,…); → [0,n] ⊆ len(s)
  strcpy(dst,src) → len(src) ⊆ len(dst)
A directed graph representing the constrain system is constructed:

- **Vertices:** the variables \((\text{len}(s), \text{alloc}(s))\)
- **Edges:** the constraints (representing the functions)

1: char* src = “testtesttest”;
2: char* dst = malloc(8*sizeof(char));
3: strcpy(dst,src);

```
[13,13] ⊆ len(src)
len(src) ↦

[8,8] ⊆ alloc(dst)
alloc(dst) ↦

[13,13] ⊆ alloc(src)
alloc(src) ↦

len(src) ⊆ len(dst) ↦
len(dst) ⊆ len(dst)
```
Compiler related approaches: BOON (II)

- The constrain solving algorithm descends through the graph until all variables stopped changing → a fixpoint is found
- A potential Buffer Overrun is found if for some string s
  \[
  \max(\text{len}(s)) > \min(\text{alloc}(s))
  \]

1: char* src = “testtesttest”;
2: char* dst = malloc(8*sizeof(char));
3: strcpy(dst,src);

\[
\begin{align*}
[13,13] \subseteq \text{len}(\text{src}) \\
\text{len}(\text{src}) \\
[8,8] \subseteq \text{alloc}(\text{dst}) \\
\text{alloc}(\text{dst}) \\
[13,13] \subseteq \text{alloc}(\text{src}) \\
\text{alloc}(\text{src}) \\
\text{len}(\text{src}) \subseteq \text{len}(\text{dst}) \\
[0,0] \subseteq \text{len}(\text{dst}) \\
\text{len}(\text{dst})
\end{align*}
\]
Compiler related approaches: CQUAL

CQUAL (2001)
- Inspired by Perl’s “tainted” mode
- Detects format string vulnerabilities
- Uses an extension to the language’s type system

How it works:
- Introduces new type qualifiers “tainted” and “untainted”
  ```c
  untainted int i;
  int main(int argc, tainted char* argv[]);
  ```
- Type inference rules are applied to propagate the type qualifier:
  ```c
  int a;
  tainted int b;
  a = b + 2;
  → a inherits the type qualifier “tainted”
  ```
Type qualifiers induce a subtyping relationship on qualified types

- untainted is a subtype of tainted

The consequences:

- It is allowed to assign an “untainted” value to a “tainted” variable
- It is forbidden to assign a “tainted” value to a “untainted” variable

```c
void f(tainted int); untainted int a;
f(a); OK
```

```c
void g(untainted int);
tainted int b;
g(b); TYPE ERROR
```
Finding Format String vulnerabilities

- **Goal:** find data paths which allows a user controlled variable to define a format string
- **All return values of function calls that contain user input are marked as “tainted”**
- **All parameters of format string exploit suspicious functions are marked as “untainted”**

**Example:**

```c
tainted char* getenv(char* name);
int printf(untainted char* fmt, ...);
char* s;
s = getenv(“PATH”);  // s gets marked as “tainted”
printf(s);           // type error
```
A classification

Approaches

static
  - syntax orientated
  - compiler related
  - theory based

dynamic
  - detection
  - prevention

“Theory based” approaches borrow concepts from the field of theoretical computer science
A classification

Approaches

- static
  - syntax orientated
  - compiler related
  - theory based
    - finite automatons
- dynamic
  - detection
  - prevention
    - program verification
Finite Automatons: XGCC

XGCC (2002)
- XGCC uses finite automatons to track assurance of conditions
- These automatons are described in the tool’s language “metal”

Checking:
- A control flow graph is generated from the source code
- The tool walks through the graph (using depth first search)
- The statements in the vertices of the cfg are examined
  - Creation of an automation
  - Triggering a transition of the automaton → the state of the automaton is updated
- If a branch (e.g. an if-statement) in the graph is encountered all automaton which are affected by the branch’s body are duplicated
Example: double free errors

- Declare any pointer v
- v.initialized
- v.freed
- free(v)
- *v
- error: double free
- error: illegal access
Finite Automatons: XGCC (III)

<table>
<thead>
<tr>
<th>Code</th>
<th>Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *v;</td>
<td>v.initialized</td>
</tr>
<tr>
<td>v = malloc(3*sizeof(int));</td>
<td></td>
</tr>
<tr>
<td>*v = 3;</td>
<td></td>
</tr>
<tr>
<td>free(v);</td>
<td>v.freed</td>
</tr>
<tr>
<td>if (...){</td>
<td></td>
</tr>
<tr>
<td>*v = 52;</td>
<td>v1.error_illegal_access</td>
</tr>
<tr>
<td>} else {</td>
<td></td>
</tr>
<tr>
<td>free(v);</td>
<td>v2.error_double_free</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Finite Automatons: MOPS

MOdel checking Programs for Security properties (2002)

- MOPS uses a comparable approach to XGCC
- For every checked property an FSA (finite state automaton) is constructed
- The program is compiled into an PDA (push down automaton)
- The tool checks if the FSA and the PDA intersect using automaton algebra

Properties of MOPS:

- Control flow sensitive
- Sound
  - If the automatons are constructed correctly, MOPS is able to guarantee the absence of specified security problems

Problem: constructing automatons is nontrivial

- Unlike XGCC MOPS is unaware of program language specific properties (like pointer initialization)
A classification

Approaches

static
  - syntax orientated
  - compiler related
  - theory based
    - finite automatons
  - program verification
dynamic
  - detection
  - prevention
Program verification

- preconditions $r$
- postconditions $e$
- Program $P = s_1; s_2; s_3; \ldots; s_n$

\{ $r$ \} $P$ \{ $e$ \}

Security tools only define pre- and postconditions for special constructs
- e.g. unsafe library functions
Splint

Splint (LCLint) (2001)
- Uses pre- and post-conditions for detecting Buffer Overflows
- Supports four types of constraints:
  - maxSet, minSet: allocated space of a buffer
  - maxRead, minRead: used space of a buffer
- Pre- and post-conditions have to be added by the programmer:
  /*@requires maxSet(dest) > maxRead(src)@*/
- Splint is able to deduct postconditions for known code constructs:
  char buf[42];
  \[\text{ensures minSet(buf) = 0 and maxSet(buf) = 41}\]
- The analysis:
  - A control flow graph is generated from the source code
  - Following the graph, Splint checks, if the pre-conditions can be met
  - If a precondition, that can’t be verified, is found, the tool emits a warning
Eau Claire (2002)
- Uses a theorem prover: “Simplify”
- Two step translation process
  - C code to Guarded Commands
  - Guarded Commands to verification condition
- “Guarded Commands” are (roughly) a translation of an instruction into its pre- and post-conditions
- Function calls are translated into the function’s pre- and post-conditions
- Annotated library functions are used for the analysis
Example:

```c
/*
spec strcpy(cpTo, cpFrom)
{
    requires $valid(cpTo):
        "the first argument must be a valid pointer"
    requires $string(cpFrom):
        "the second argument must be a valid string"
    requires $length(cpTo) > $string_length(cpFrom):
        "the array must be large enough to hold the entire string"

    modifies $elems(cpTo)

    ensures forall(i) ((i >= 0 && i <= $string_length(cpFrom)) implies
        $final(cpTo[i]) == cpFrom[i])

    // next item is true but not necessary for the spec
    //ensures $string_length(cpTo) == $string_length(cpFrom)
}
*/
void strcpy(char* cpTo, char* cpFrom);
```
A classification

- Approaches
  - static
  - dynamic

Commercial tools
Commercial Tools: Fortify

Fortify Source Code Analysis Engine

- Four modules:
  - Data Flow Analyzer
  - Semantic Analyzer
  - Control Flow Analyzer
  - Configuration Analyzer

- Multiple languages are supported
  - C, C++, Java, JSP, PL/SQL, C#, XML

- Supports custom “Rulepacks”

- Provides IDE Plug-Ins
  - Borland JBuilder
  - Eclipse
  - MS Visual Studio

Commercial Tools: Prexis

Ounce Labs Prexis

- Two modules
  - CAM++: C/C++ Assessment Module
  - JAM: Java Assessment Module
- Context Sensitive Analysis
- “Multi Dimensional Method”
  - Some code regions are checked more thoroughly than others
- Checks for:
- Only loosely integrated in IDEs

http://www.ouncelabs.com/overview.html
Commercial Tools: Coverity

Coverity Prevent and Coverity Extend

- Based on XGCC
- “Prevent” enforces predefined conditions to eliminate known security problems
- “Extend” is a tool to add custom checks to the process
- Also checks for not security relevant errors
- Integrates in IDEs
- Provides source code Browser

http://www.coverity.com
Commercial Tools: CodeAssure

Secure Software CodeAssure Workbench
- Assessment of control and data flows
  - Integer and string ranges
  - Function calls
  - Aliases
- Knowledge:
  - 40 types of vulnerabilities
  - Thousands of individual examinations and rules
- Provides severity levels
- Language Packs:
  - Java
  - C
  - C++
- Integrated in Eclipse

http://www.securesoftware.com/products/source.html
Application Defense Developer

- Supports 13 languages:
  - C, C++, C#, VBScript, VBA, ASP, Jscript, JavaScript, PHP, Python, LISP, ColdFusion, Perl
- Time to check the code < 1 minute
- "proprietary artificial intelligence engine"
- XML Output
- Provides IDE
- Product only available in combination with the company’s other services

http://applicationdefense.com/ApplicationDefense_Products.htm#developer
Commercial Tools: SPI Dynamics

SPI Dynamics DevInspect and SecureObjects

- Integrates in Visual Studio.NET
  - Only checks languages, that are part of the .NET framework
- Focuses on Web Applications
  - Input validation
  - XSS
  - SQL injection
- Provides code fragments

A classification

Approaches

- static
  - syntax orientated
  - compiler related
  - theory based
- dynamic
  - detection
  - prevention

A comparison of selected static tools
A comparison

- A comparison test was carried through by J. Wilander and M. Kamkar at the University of Linköping, SE (2002)
- The tested tool were: Flawfinder, ITS4, RATS, Splint and BOON
- For the test a uniform testfile was created, which included:
  - 15 unsafe buffer writings
  - 13 safe buffer writings
  - 8 unsafe format string calls
  - 8 safe format string calls

|                      | Flawfinder | ITS4 | RATS | Splint | BOON *
|----------------------|------------|------|------|--------|--------
| True Positives       | 22 (96%)   | 21 (91%) | 19 (83%) | 7 (30%) | 4 (27%) |
| False Positives      | 15 (71%)   | 11 (52%) | 14 (67%) | 4 (19%) | 4 (31%) |
| True Negatives       | 6 (29%)    | 10 (48%) | 7 (33%) | 17 (81%) | 9 (69%) |
| False Negatives      | 1 (4%)     | 2 (9%)  | 4 (17%) | 16 (70%) | 11 (73%) |
## A comparison (II)

<table>
<thead>
<tr>
<th>Vulnerable Function</th>
<th>Flawfinder</th>
<th>ITS4</th>
<th>RATS</th>
<th>Splint</th>
<th>BOON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>gets()</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>scanf()</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>fscanf()</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sscanf()</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>vscanf()</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>vsscanf()</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>vfscanf()</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>cuserid()</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>sprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>strcat()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>strcpy()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>strlen()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vsprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>strtrans()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>printf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>snprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vfprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vsprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vfprintf()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Dealing with false positives

- It is a given that there will always be a certain ratio of false positives

- Possibilities to handle this problem:
  - Improving the tools to lower the ratio
  - Giving up soundness (danger: false negatives)
  - Weighting the results
FLF: Front Line Functions (2003)

- Hypothesis: The closer a function is to (user) input, the more likely it contains an exploitable vulnerability

- The program is examined:
  - Functions that receive user input are labeled as *Inputs*
  - Functions with potential vulnerabilities are labeled as *Targets*
  - Furthermore a call graph of the program is generated

- To weight a given Target, the *FLF density* $k$ for each Input is calculated
  - $k = \frac{p}{m}$ with
    - $p =$ maximal number of functions on the call graph between the Input and the Target
    - $m =$ total number of functions in the program
  - The largest $k$ is chosen as the FLF density
A classification

- Approaches
  - static
    - syntax orientated
  - dynamic
    - compiler related
    - theory based
    - detection
    - prevention

Dynamic approaches
Dynamic approaches

- Dynamic approaches work on runtime
- During program execution buffer overflows are detected or prevented
- To achieve this goal extra code is included in the watched program
  - Before compilation: Additional C code is added to the source
  - During compilation:
    - The compilation process is altered
    - Certain statement or mechanisms get translated differently (compared to standard compilation)

Note:

- In this presentation only tools that check for (stack) buffer overflows are discussed
- Approaches that look for format string exploits or heap corruption exist as well
- (check out Yves Younan’s talk)
dynamic versus static approaches

- Sophisticated static tools try to determine runtime conditions of programs before execution
  - Control flow
  - Loop heuristics
  - Data flow
- Dynamic tools evaluate data that occurs when the program is actually run
  - Therefore the tools have no need for approximation/guessing of runtime behavior
A classification

- Approaches
  - static
    - syntax orientated
    - compiler related
    - theory based
  - dynamic
    - detection
    - prevention
    - finite automatons
    - program verification
Dynamic detection: STOBO


- Dynamic extension of BOON’s approach
- Tracks possible buffer length during execution
- The tool is supposed to accompany program testing

How it works:

- The source code is altered before compilation
- The tool keeps track of all buffer sizes in a global table
  - For every variable declaration that declares a buffer a special function call is added
  - All functions and constructs that allocate buffer memory are wrapped
- All functions which could lead to buffer overflows are wrapped
- If a wrapped function detects a potential overflow a warning gets generated
Dynamic detection: STOBO (II)

**Source code**

```c
char buf[100];

char *ptr;
ptr = malloc(20);

strcpy(ptr,buf);
```

**STOBO output**

```c
char buf[100];
__STOBO_stack_buf(buf, sizeof(buf));

char *ptr;
ptr = __STOBO_const_mem_malloc(20);

__STOBO_strcpy(ptr,buf);
```

STOBO is able to notice the possible buffer overflow
A classification

Approaches

- static
  - syntax orientated
  - compiler related
  - theory based

- dynamic
  - detection
  - prevention
    - finite automatons
    - program verification
    - bounds checker
    - stackguards
Stack protectors

- Stack protectors try to prevent exploitation by altering the underlining program semantics
  - Enhanced function-prolog and -epilogue
  - Separate stack for return addresses
  - Reordering of local variables
Stack protectors: Canaries

Protection of the return address through a canary value

- The function prologue and epilogue get enhanced on compile time
- The function prologue adds a canary value on the stack before the return address
- If a vulnerable buffer is overflown in order to overwrite the return address, the canary value is overwritten as well
- The function epilogue checks, if the canary value is unaltered, before the return from the function is executed
Stack protectors: Canaries (II)

```
int something(int para){
  char buf[8];
  ...
  strcpy(buf, "shellcodeshellcode");
}
```
Stack protectors: Canaries (II)

int something(int para) {
    char buf[8];
    ...
    strcpy(buf, "shellcodeshellcode");
}

function parameter
return address
canary value
saved FP
shel

FP →
SP →

Higher addresses

Lower addresses
Stack protectors: Canaries (II)

Higher addresses

```c
int something(int para) {
    char buf[8];
    ...  
    strcpy(buf, "shellcodeshellcode");
}
```

Lower addresses
Stack protectors: Canaries (II)

Higher addresses

```
int something(int para){
    char buf[8];
    ...
    strcpy(buf, "shellcodeshellcode");
}
```

Lower addresses

Function parameter
Return address
Canary value
Stack protectors: Canaries (II)

int something(int para){
    char buf[8];
    ...
    strcpy(buf, "shellcodeshellcode");
}

- The canary value gets overwritten.
- The attack is detected
Stack protectors: Canaries (III)

Types of canaries:
- Random canary
- Random XOR canary
- Null-canary ("\0\0\0\0")
- Terminator canary ("\0\n\r\ff")

Limitations
- Stack canaries protect only against Buffer Overflow that try to overwrite the return address
- No protection against
  - Heap overflows
  - Formatstring exploits
  - Function pointer overwriting
  - Alteration of local variables
Stack protectors: Canaries (IV)

Tools that use stack canaries:
- StackGuard
- Microsoft VisualStudio.NET
- ProPolice/GCC (with some enhancements)

Additional problem
- When a canary violation is detected, the program usually exits
- This turns the vulnerability into a denial of service opportunity
Stack protectors: ProPolice SSP

ProPolice Stack-Smashing Protection (2000) by IBM Research
- Uses stack canary values
- Additionally ProPolice reorders the values on the stack

| ... |
| function arguments |
| return address |
| saved frame pointer |
| guard value |
| local buffers |
| local variables and pointers |
| ... |
Stack protectors: ProPolice SSP (II)

- Vulnerable function arguments are protected through local copies

```c
void bar(void (*func)()){
    char buf[128];
    ...
    strcpy(buf, getenv("HOME"));
    (*func)();
}
```

Disadvantages:
- No reordering of struct-elements
- No protection of pointer arrays
Stack protectors: Stack Shield

Stack Shield (2001)
- Has two options for return address protection
- Global Ret Stack
  - An additional global stack containing the return address is maintained
  - In the function prologue the return address is written on the global stack
  - In the function epilogue the saved return address is copied back onto the stack
  - No detection of stack smashes, the program continues to run
- Ret Range Check
  - Using a global variable, the beginning address of the .data segment is calculated
  - Every return value is compared with this address
  - If the return address is smaller it points to the .text segment and is therefore probably safe
  - This approach is vulnerable against return-to-libC attacks
A classification

Approaches

Static
- Syntax orientated
- Compiler related
- Theory based

Dynamic
- Detection
- Prevention
  - Finite automatons
  - Program verification
  - Bounds checker
  - Stack protectors
Bounds checker


- CRED uses an global object table to track memory regions
- For every memory reference that is created during program execution an entry in the object table is created
- An entry consists of a base address and the amount of allocated memory
- Whenever a pointer is dereferenced, the object table is used to determine, if the access is correct
- To achieve this\texttt{malloc()}, \texttt{free()} and code that dereferences pointers is replaced
- Out of bounds pointers are legal, as long they aren’t dereferenced
- With this method all memory regions are protected against overflows, not only stack buffers
- Memory regions that were allocated by external libraries aren’t protected
- The (experimental) determined runtime overhead is between 26\% and 150\%
A classification

Approaches

- static
- dynamic

Combining static and dynamic techniques
Combining static and dynamic approaches

CCured (2002)
- Combines type inference and runtime checking
- Pointers are separated according to their usage
  - SEQ: Sequence pointers – pointer may be subject to pointer arithmetic
  - SAFE: Pointer remains unaltered on program execution
  - DYNAMIC: Pointer may be subject to type casting
- Code handling “unsafe” pointers is instrumented with runtime checks
A classification

- Approaches
  - static
  - dynamic

Conclusion
Conclusion

- Current limitations are unknown
  - Which kind of vulnerabilities are detectable today?
  - Which kind of vulnerabilities are still not covered?
- What is the actual ratio between real vulnerabilities and false positives
  - How should be dealt with potential false positives?
- Static tools
  - Ignorance of C++ (and/or object orientated programming)
  - Narrow focus (bounds checking, format strings, toctou,...)
  - To check for all potential security flaws, we would have to combine different approaches
    → More false positives
- Dynamic tools
  - Only defend against known attack vectors
  - If new/different ways of exploitation are discovered, many dynamic tools are helpless
The end

Thanks for listening…
Bibliography (static tools)


Bibliography (dynamic tools)


- **StackShield**: http://www.angelfire.com/sk/stackshield/


Bibliography (Comparisons)


