# Secure Development with Static Analysis

Ted Unangst
Coverity

# Source Code Analysis: Why do you care?

- As a developer, source code is the malleable object
- Many techniques can be shared between source and binary analysis
- Flip side: Improve RE by looking for blind spots in source analysis

# Outline

- Program Analysis
- Source Code analysis
- Source code analysis for security auditing
- Deployment

# Program Analysis

- Source vs Binary
- Static vs Dynamic
- Complementary

# Source and Binary analysis

- Source
  - Easy to identify location of flaw
  - CPU independent
  - Language depen.
  - Environment independent?
  - 1st party only
- Binary
  - Hard to map back to source
  - CPU dependent
  - Language indep?
  - Environment independent?
  - 3rd party utility

# Static and Dynamic Analysis

- Static
  - Complete coverage
  - false positives
  - Can analyze anytime, anywhere
  - Precise description of problem, unknown impact
- Dynamic
  - Very rare to get close to 100%
  - No false positives
  - Requires ability to run program
  - Precise understanding of impact, possibly unknown cause

# Static Source Code Analysis

- The source has a lot of knowledge embedded
  - Extract the good parts, ignore the remainder
- Many types of analysis
  - Different ways to approach the problem
  - Different goals
    - Find Defects
    - Enhance Run time analysis
    - Gain insight into code

# Static Source Code Analysis

- Type qualifier checking
- Pattern recognition
- Source rewriting
- Model checking
- Simulated execution
  - Flow sensitive?
  - Context sensitive?

# Simulated Execution

- Flow sensitive

```
bar() { free(g_p); }
baz() { free(g_p); }
foo() { if (x) bar(); else baz(); }
```

- Context sensitive

```
if (use_malloc)
  p = malloc();
/* … */
if (use_malloc)
  free(p);
```

- How much state to track?
  - Exponential number of paths
  - Loops
  - Heap is unbounded

# Advantages

- Automatic
  - Should be capable of scanning the entire source base on a regular basis
  - Infrastructure should be able to adapt to changing code bases
- Complete
  - Looks at all the code including edge conditions and error handling

# Challenges

- Parsing
  - Any non-trivial analysis requires parsing the code
  - Nobody writes standard C; they write [name of compiler] C
  - Many extensions are not documented (if they're even unintentional)
  - Hard to analyze if you can't read it

# Parsing Fun

```
struct s { // parenthesis what?
     unsigned int *array0[3 ];
   ()unsigned int *array1[3 ];
   ( unsigned int *array2[3)];
   ( unsigned int *array3[3 ];
};
{ // lvalue cast?
    const int x = 4;
    (int)x = 2;
}
{ // what's a static cast?
    static int x;
    static char *p =
     (static int *)&x;
}
{ // no, that's not =
   int reg @ 0x1234abcd;
}
```

```
{ // goto over initializer
   goto LABEL;
   string s;
LABEL:
   cout << s << endl;
}
{ // bad C. but in C++?
   char *c = 42;
}
{ // init struct with int?
   struct s {
        int x; int y;
   };
   struct s z = 0;
}
{ // new constant types
   int binary = 11010101b;
}
```

# Challenges

- What properties to look for?
- Can only analyze what we can see
- Linkage affects run time behavior
- RPC, COM, function pointers
- False positives
- False negatives

# Test generation

- Use source code analysis to find edge cases faster
- Precisely directed test cases

# A window into the black box

- Occasionally we have some of the source
- XML, JPEG, PNG libraries
- Use source code analysis to discover properties about the API
  - Which functions allocate memory? Free memory?
  - Write to a buffer? How big?

# Security

- Static Source Analysis pros
  - Thorough
  - Reveals root cause and path
  - data flow tracking (?)
- Cons
  - Unable to understand impact
  - Some data dependencies are just too complicated

# Security

- What qualifies as a defect?
  - Every strcpy()?
- What properties can we determine statically?
  - Buffer overruns
  - Integer overflows
  - Race conditions
  - Memory leaks

# SQL Injection Example

```
name = read_form_entry("NAME");
res = run_query("select * from user
  where name = '%s'", name);
```

- How do we know `name` is bad?
- How do we know `run_query` will behave incorrectly?
- Configuration:
  - `read_form_entry : USERDATA(RETURN)`
  - `run_query : TRUST(ALLARGS)`

# SQL Injection Example

```
csv = read_form_entries();
otherval = p = strchr(csv, ‘,’);
*p++ = 0;
name = p;
p = strchr(p, ‘,’);
*p++ = 0;
res = run_query(“select * from user where
  name = ‘%s’”, name);
```

- Configuration:
  - read_form_entry : USERDATA(RETURN)
  - run_query : TRUST(ALLARGS)
  - strchr : COPY(ARG0, RETURN)

# SQL Injection Example

```
csv = read_form_entries();
otherval = p = strchr(csv, ‘,’);
*p++ = 0; name = p;
p = strchr(p, ‘,’); *p++ = 0;
name = escape_sql(name);
res = run_query(“select * from user where
  name = ‘%s’”, name);
```

- Configuration:
  - read_form_entry : USERDATA(RETURN)
  - run_query : TRUST(ALLARGS)
  - strchr : COPY(ARG0, RETURN)
  - escape_sql : OKDATA(RETURN)

# SQL Injection Example

```
csv = read_form_entries();
otherval = p = strchr(csv, ‘,’);
*p++ = 0; name = p;
p = strchr(p, ‘,’); *p++ = 0;
if (!validate_name(name))
   return (EINVAL);
res = run_query(“select * from user where name =
   ‘%s’”, name);
```

- Configuration:
  - read_form_entry : USERDATA(RETURN)
  - run_query : TRUST(ALLARGS)
  - strchr : COPY(ARG0, RETURN)
  - validate_name : 0(RETURN) => USERDATA(ARG0);
    1(RETURN) => OKDATA(ARG0)

# Configuration

- Tedious to annotate by hand
- Can use statistics to derive correct function pairings
  - Some developers may get it wrong more than right ☹
- Start at read, recv, ...
  - Assume return values are tainted
  - Assume all functions trust input
  - Converges quickly
- We haven't verified `escape_sql` works correctly

# Exploit?

```
if (issetugid() == 0)
    errx(1, “Improperly installed”);

str = getenv(“NUMTHREADS”);
n = atoi(str);
n *= sizeof(widget);
ptr = malloc(n);
```

# Soundness

- Many tools cut corners
  - Pointer analysis is hard
  - Two choices:  leave some bugs behind or get swamped by false positives
  - Delicate balance
  - Still very good at catching the low hanging fruit and finding dangerous constructs

# Cost of False Positives and False Negatives

- False positives are costly
  - If uncontrolled, can easily sap more time from development or auditing than the analysis saves
  - Over time, the trend is to 100% false positives
  - Important consideration for tool adoption
- Cost of false negatives is harder to estimate
  - False sense of security
  - If you start spending less time testing, you're in for a nasty surprise
  - Analysis should help focus and direct audits, not replace them
  - Some properties can be verified; in the general case it's impossible

# Build or Buy?

- Build
  - Find those bugs you are especially interested
  - Hard, Hard, Hard
  - Users are never happy
  - How much do 5-10 developers cost per year?
- Buy
  - Maybe Not a perfect fit
  - Checks many additional properties
  - Generality allows migration to other projects

# Understanding the Tool

- The average analysis tool doesn't think like a developer
- Error messages may require interpretation
  - "What do you mean 'n' could be 4294967295?"
  - Too much information to present all of it; what can be omitted?
  - Best solution is regular exposure

# Usage (Care and feeding of your source code analysis tool)

- The tool is static; your usage should not be
- Most effective with regular usage
  - You can't fix everything the week before the release
- Adapt checkers to unique problems
- Simplify code where possible to eliminate false positives
  - But don't try to outsmart the tool

# Organizational Deployment

- Developers are not always incentivized to use the tools available
  - More work
  - Accountability!
  - Needs an internal champion
- Maintenance
  - Verify all code going out the door is being checked

# Round up

- Static source code analysis can augment other forms of analysis
- Mostly confined to developers (need source) but adoption is slow or lacking in many organizations
- Much like secure programming, performing security analysis requires dedication and patience

# The End

- Thanks
  - RECon
  - Coverity
- Questions?