Towards practical reactive security audit using extended static checkers[1]

Julien Vanegue [1] Shuvendu K. Lahiri [2]

[1]Bloomberg LP, New York

[2]Microsoft Research, Redmond

May 20, 2013

[1]The work was conducted while the first author was employed at Microsoft.

Problem: Find software security vulnerabilities in legacy applications

- **Legacy applications**
  - Operating system applications
  - Core drivers and other kernel components
  - Browser renderer and browser management libraries
  - COM components accesses
- **Security vulnerabilities**
  - (Heap) Buffer overruns
  - Double frees
  - Use-after-free
    1. Reference counting issues
    2. Dangling pointers
  - Information disclosures
  - Dynamic type confusions
  - Zero allocations
  - Untrusted component execution (e.g. dll/ocx loading)

Problem: Find software security vulnerabilities in legacy applications

- **Legacy applications**
  - Operating system applications
  - Core drivers and other kernel components
  - Browser renderer and browser management libraries
  - COM components accesses
- **Security vulnerabilities**
  - (Heap) Buffer overruns
  - Double frees
  - Use-after-free
    1. Reference counting issues
    2. Dangling pointers
  - Information disclosures
  - Dynamic type confusions
  - Zero allocations
  - Untrusted component execution (e.g. dll/ocx loading)

Finding all such security issues ahead of time in a cost-effective manner is infeasible.

Problem: Find software security vulnerabilities in legacy applications

- **Legacy applications**
  - Operating system applications
  - Core drivers and other kernel components
  - Browser renderer and browser management libraries
  - COM components accesses
- **Security vulnerabilities**
  - (Heap) Buffer overruns
  - Double frees
  - Use-after-free
    1. Reference counting issues
    2. Dangling pointers
  - Information disclosures
  - Dynamic type confusions
  - Zero allocations
  - Untrusted component execution (e.g. dll/ocx loading)

Finding all such security issues ahead of time in a cost-effective manner is infeasible.
Can we do better for some interesting scenarios?

The MSRC mission

The **Microsoft Security Response Center** (MSRC) identifies, monitors, resolves, and responds to security incidents and Microsoft software security vulnerabilities.

http://www.microsoft.com/security/msrc/whatwedo/updatecycle.aspx

"...The MSRC engineering team investigates the surrounding code and design and searches for other variants of that threat that could affect customers."

Ensuring there are no variants of a given threat is an **expensive** process of automated testing, manual code review, static analysis. The process is effective (*finds additional bugs*) but far from providing any assurance/coverage.

A more realistic problem

Reactive security audit

- Given an **existing security threat** (e.g. independent researcher finds an unknown vulnerability).
- Describe **variants** of the threat.
- Perform **thorough checking** of variants of the threat on the attack surface.
- In a **time constrained fashion** (must obtain results before a security bulletin is released (time frame: often days, weeks or months)).

Tools for an auditor

A **security auditor**

- Has domain knowledge to create variants
- Can guide tools to help with the static checking
- But, cannot spend months/years to perform a formal proof of a given component

Tools for an auditor

A **security auditor**

- Has domain knowledge to create variants
- Can guide tools to help with the static checking
- But, cannot spend months/years to perform a formal proof of a given component

A reactive security audit tool must be *cost-effective*:

- **Configurable** to new problem domains.
- **Scalable** (Millions of lines of code).
- **Accurately** capture the underlying language (C/C++) semantics.
- **Transparent** with respect to the reasons for failure. No black magic.

Why audit?

Other approaches find bugs, but **lack user-guided refinement and controllability**.

- **White-box fuzzing** (e.g. SAGE)
  - Not property guided (can't exhaust a million lines of code)
  - Difficult to apply to modules with complex objects as inputs
- **Model checking** (e.g. SLAM)
  - Typically state machine properties (can't distinguish objects, data structures)
  - Difficult to scale to more than 50KLOC
- **Data-flow analysis** (e.g. ESP)
  - Ad-hoc approximations of source (C/C++) semantics
  - Not suited for general property checking

In other words: A user could not easily influence the outcome of these tools, even if they desired.

Extended static checkers to the rescue

Extended static checking

- **Precise modeling of language semantics** (may make clearly-specified assumptions)
- **Accurate intraprocedural checking**
- **User-guided annotation inference**
  1. Interprocedural analysis (made easy using templates)
  2. Loop invariants (not the subject if this talk)

Made possible by using automated theorem provers.

ESC/Java[Flanagan,Leino et al. '01], HAVOC [Lahiri, Qadeer et al. '08], Frama-C ....

Contributions

- Explore the problem of reactive security audit using extended static checker **HAVOC**.
- Extensions in HAVOC (called **HAVOC-LITE**) to deal with complexity of applications (C++, million LOCs, usability, robustness).
- Case study on C++/COM components in the browser and the OS at large:
    1. Found 70+ *new* security vulnerabilities that have been fixed.
    2. Vulnerabilities were not found by other tools after running on same code base.
    3. Discussion of the effort required vs. payoff for the study.

Overview of the rest of the talk

- Overview example
- HAVOC overview
- HAVOC $\rightarrow$ HAVOC-LITE
- Case study
- Conclusions

Overview example (condensed)

```c
typedef struct tagVARIANT
{
#define VT_UNKNOWN   0
#define VT_DISPATCH  1
#define VT_ARRAY     2
#define VT_BYREF     4
#define VT_UI1       8
 (...)
  VARTYPE vt;
  union {
      ...
      IUnknown   *unk;
      IDispatch  *disp;
      SAFEARRAY  *parray;
      BYTE       *pbVal;
      PVOID      byref;
      ...
  };
} VARIANT;
```

```c
void t1bad() {
  VARIANT v;
  v.vt = VT_ARRAY;
  v.pbVal = 0;
}
void t2good() {
  VARIANT v;
  v.vt = VT_BYREF | VT_UI1;
  use_vfield(&v);
}
void use_vfield(VARIANT *v)
  v->pbVal = 0;
}
void t2good2() {
  VARIANT v;
  set_vt(&v); v.pbVal = 0;
}
void set_vt(VARIANT *v) {
  v->vt = VT_BYREF|VT_UI1;
}
```

Overview example: Annotations

```
//Field instrumentations (Encoding the property)

__requires(v->vt == VT_ARRAY)
__instrument_write_pre(v->parray)
void __instrument_write_array(VARIANT *v);

__requires(v->vt == (VT_BYREF|VT_UI1))
__instrument_write_pre(v->pbVal)
void __instrument_write_pbval(VARIANT *v);
```

Overview example: Annotations

```
//Field instrumentations (Encoding the property)

__requires(v->vt == VT_ARRAY)
__instrument_write_pre(v->parray)
void __instrument_write_array(VARIANT *v);

__requires(v->vt == (VT_BYREF|VT_UI1))
__instrument_write_pre(v->pbVal)
void __instrument_write_pbval(VARIANT *v);
```

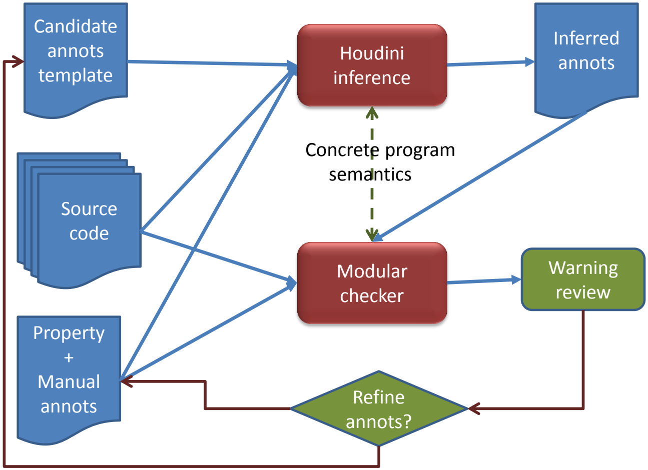
```
//Func instrumentations with candidates (Encoding inference)

__cand_requires(v->vt == (VT_BYREF|VT_UI1))
__cand_requires(v->vt == VT_ARRAY)
__cand_ensures(v->vt == (VT_BYREF|VT_UI1))
__cand_ensures(v->vt == VT_ARRAY)
__instrument_universal_type(v)
__instrument_universal_include("*")
void __instrument_candvariant(VARIANT *v);
```

Overview example: Tool output

- Addition of preconditions/postconditions on relevant methods and fields
  - E.g. adds an assertion before access to `pbVal` field in each of the procedures
- Inferred preconditions and postconditions
  - E.g. precondition `__requires(v->vt == (VT_BYREF | VT_UI1))` on `func_use_vfield`
  - E.g. postcondition `__ensures(v->vt == (VT_BYREF | VT_UI1))` on `func_set_vt`
- Warning only on `t1bad` procedure.

HAVOC flow

The modular checker uses Boogie program verifier and Z3 prover.



HAVOC $\rightarrow$ HAVOC-LITE

- **Supporting C++** constructs used commonly in the COM applications.
- **Scalable interprocedural inference**.
- **New instrumention** mechanisms to deal with classes.
- **Usability and robustness** enhancements.

C++ support

Extend the memory model of HAVOC to support:

- Classes and attributes.
- C++ references.
- Casts (static and dynamic).
- Constructors, destructors and instance methods.
- C++ operators.
- Method overloading.
- Multiple inheritance.
- Dynamic dispatch and type look-up.

Ability to **annotate** instance methods, attributes and operators.
Extend **instrumentations** to instrument all methods of a class, or a method in all classes.

Scalable annotation inference

- HAVOC used Houdini[Flanagan, Leino '01] algorithm for inferring inductive annotations from a set of candidates.
- Previous approach only worked for a few hundred to thousand methods in a module (memory blowup)
- A **two-level Houdini algorithm** is used
    1. Break inference into a large number of smaller calls to Houdini (see paper).
    2. Each call to Houdini consists of few hundred methods only.
    3. Different Houdini calls can act on same methods.
    4. Algorithm runs until the set of contracts for methds does not change anymore.

Case study

- Properties checked
- Results
- Example bug
- Cost-effectiveness

Checked security properties

Properties	Description
Zero-sized allocations	Dynamic memory allocations never of size 0
Empty array ctors	VLA allocations never of size 0
VARIANT initialization	VARIANT structures must be initialized before use
VARIANT type safety	VARIANT fields usage consistent with run-time type
Interface refcounting	Interfaces must not be released without prior ref/init
Library path validation	Run-time modules always loaded with qualified path
DOM info disclosure	DOM accessors return failure on incomplete operations

These properties are variants of vulnerabilities reported in a large set of MSRC bulletins (see paper).

Results : Found vulnerabilities

Prop	LOC	Proc #	Vulns	Checktime	Inference
Zero-sized allocations	2.8M	58K	9	3h14	3h22
Empty array ctors	1.2M	3.1K	0	26m	6m13
VARIANT initialization	6.5M	196K	5	5h03	11h40
VARIANT type safety	6.5M	196K	8	5h03	11h40
Interface refcounting	2M	11.2K	4	2h26	20h
Library path qualification	20M	Mils	35	5d	N/A
DOM info disclosure	2.5M	100s	2	1h42	N/A

An additional 7 bugs were found for **Probe validation** of userland-pointers.

Inference measurements

Properties	Warns Warns	Warns with inf	Improv.	Cand.	Inf. cand.
Zero-sized allocations	71	50	29%	75162	42160
Empty array constructor	45	35	22%	4024	446
VARIANT initialization	216	117	45%	100924	770
VARIANT type safety	83	68	18%	100924	770
Interface refcounting	746	672 (3)	10%	234K	1671
DOM info disclosure	82	N/A	N/A	N/A	N/A
Library path validation	280	N/A	N/A	N/A	N/A

The instrumentation + candidates took less than 100 lines for each property.

Interpretation : Inference is a useful to reduce the number the false positives across applicable properties.

Example bug: COM VT VARIANT vulnerability

```
HRESULT CBrowserOp::Invoke(DISPID dispId, DISPPARAMS *dp)
{
  switch (dispId) {
    case DISPID_ONPROCESSINGCOMPLETE:
      if (!dp || !dp->rgvarg || !dp->rgvarg[0].pdispVal) {
        return E_INVALIDARG;
      }
      else {
        IUnknown *pUnk = dp->rgvarg[0].pdispVal;
        INeededInterface *pRes = NULL;
        hr = pUnk->QueryInterface(IID_NeededIface, &pRes);
        if (hr == S_OK) {
          PerformAction(pRes, dp);
          ReleaseIface(pRes);
        }
      }
    break;
    default: return DISP_E_MEMBERNOTFOUND;
  }
  return S_OK;
}
```

Cost-effectiveness of HAVOC-LITE

- Complement existing techniques when domain specific knowledge is required.
    1. Allow broader coverage than fuzzing or testing.
    2. Does not replace fuzzing or manual audit, but complement them.
- Review of results and creation of new annotations/instrumentations determine the level of warnings.
- In our experience, cost-effectiveness is best when warnings # stay smaller than 100 per Million LOC.
    - Easier to review than trying to add more inference

Conclusion

- HAVOC-LITE enables practical reactive vulnerability checking on large C/C++/COM software.
- Explaining the reason of warnings is key to wider usability [Lahiri, Vanegue, VMCAI'11]
- Future work
  - Currently attempting to roll out to other security auditors and properties.
  - Work inspired new methods for assigning confidence to warnings [Blackshear, Lahiri PLDI'13]
  - Combine with property-directed inlining [Lal,Lahiri,Qadeer CAV'12]

The end

Questions? Thank you.