On the Limits of Information Flow Techniques for Malware Analysis and Containment

Lorenzo Cavallaro[1] Prateek Saxena[2] **R. Sekar**[3]

Department of Computer Science, UC Santa Barbara[1]
Department of Computer Science, UC Berkeley[2]
Department of Computer Science, Stony Brook University[3]

GI SIG SIDAR Conference on
Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)

10-11 July, 2008

Static Information Flow Analysis

- Determines whether the value of a variable x is influenced by the value of another variable y
- Typically based on *non-interference:* Changes to a *sensitive* variable y should not result in changes to a *public* variable x

- Information flow literature dominated by static analysis
  - Purely dynamic analysis techniques cannot capture non-interference
  - Operate on type-safe high-level languages
- Static analysis is difficult on binaries — especially on malware, which often employs obfuscation techniques
  - Even disassembly is hard.
- Result: techniques that operate on COTS software typically use dynamic analysis

Dynamic Information Flow Analysis

... or Taint Analysis in a Nutshell

Determines, at runtime, whether a variable x is influenced by another variable y

- Track how a program's *untrusted* data (input) *flows* into security-sensitive *sinks*
  - $x := y$ (explicit data-dependent flow)
  - **if** $y = k$ **then** $x = k^{'}$ (explicit control-dependent flow)
  - Implicit flows are not handled.
    $x = 0;$
    if $y = 1$ **then** $x = 1$
    Note: x has no control dependence on y when $y = 0$

↑ Enforce security policies on sinks to detect improper usage of *tainted* data

Dynamic Information Flow Analysis

... or Taint Analysis in a Nutshell

Determines, at runtime, whether a variable x is influenced by another variable y

- Track how a program's *untrusted* data (input) *flows* into security-sensitive *sinks*
  - $x := y$ (explicit data-dependent flow)
  - **if** $y = k$ **then** $x = k^{'}$ (explicit control-dependent flow)
  - Implicit flows are not handled.
    $x = 0$;
    if $y = 1$ **then** $x = 1$
    Note: x has no control dependence on y when $y = 0$

↑ Enforce security policies on sinks to detect improper usage of *tainted* data

On the Limits of Information Flow Techniques

Motivation

Dynamic information-flow techniques have been used in the context of

- Benign applications
  - Memory errors
  - Command and SQL injection, Cross-Site Scripting, ...
- *Untrusted* (i.e., potentially malicious) applications. Examples:
  - To detect remote control bot-like behavior
  - To discover trigger-based (malicious) behaviors
  - To detect plug-ins run-time violation of policies

$\Rightarrow$ Subjected to a slew of evasion techniques, as we'll show in this talk

On the Limits of Information Flow Techniques

Motivation

Dynamic information-flow techniques have been used in the context of

- Benign applications
  - Memory errors
  - Command and SQL injection, Cross-Site Scripting, . . .
- *Untrusted* (i.e., potentially malicious) applications. Examples:
  - To detect remote control bot-like behavior
  - To discover trigger-based (malicious) behaviors
  - To detect plug-ins run-time violation of policies

  $\Rightarrow$ Subjected to a slew of evasion techniques, as we'll show in this talk

Information Flow for Malware Analysis/Containment I

Detecting remote control bot-like behavior

Stinson *et al.* suggested a dynamic information flow technique for detecting "remote control" behavior

- Bots receive commands from a central site and carry them out

⇒ Manifestation of a *flow* of information from an input operation to an output operation

- Implementation relied on *content-based* tainting, which is easily evaded (as noted by Stinson et al)

↓ What we show: *malware can easily defeat any dynamic taint-tracking implementation*

Information Flow for Malware Analysis/Containment II

Analyzing Run-time Behavior of Shared-Memory Extensions

Egele *et al.* suggested a dynamic information flow for tracking the flow of confidential data as processed by web browser and Browser Helper Objects (BHOs)

- The actions of BHOs loaded in the address space (AS) of the browser are monitored
- Needs to distinguish the *execution contexts*, i.e., proper and improper use of tainted or sensitive data
  - As used by the browser itself
  - As used by the BHOs on their own
  - As used by the browser on behalf of the BHOs

↓ What we show: *New attacks that (a) involve BHO corruption of browser data, (b) confuse attribution, or (c) evade taint-tracking mechanisms*

Information Flow for Malware Analysis/Containment III

Analyzing Future Behavior of Malware

Moser *et al.* suggested a dynamic information flow technique to discover malware behaviors by exploring execution paths

- Taints trigger-related inputs (e.g., calls to obtain time, network reads)
- Dynamic taint-tracking exploited to discover input-dependent conditionals
- Use a decision procedure to generate values for program variables that can result in execution of untaken branch

↓ What we show: *Memory errors can embedded in malware to prevent discovery of input-dependent branches*

Outline

Outline

Evasion Using Control Dependence

```
char y[256], x[256];
...
int n = read(network, y, sizeof(y));
for (int i=0; i < n; i++) {
  switch (y[i]) {
    case   0: x[i] = (char)13; break;
    case   1: x[i] = (char)14; break;
    ...
    case 255: x[i] = (char)12; break;
    default: break;
  }
}
```

- `y` gets copied into `x` even though there is no *explicit* direct assignment between them

Evasion Using Covert Channels

Implicit Flows: Copying an arbitrary quantity of data

```
void memcpy(u_char *dst, const u_char *src, size_t n) {
   u_char tmp;

   for (int i = 0; i < n; i++) {
      for (u_char j = 0; j < 256; j++) {
         tmp = 1;
         if (src[i] != j) {
            tmp = 0;
         }
         if (tmp == 1) {
            dst[i] = j;
         }
      }
    }
}
```

Implications

- Increase of false positives if control-dependences are tracked
  $\Rightarrow$ Diminish the ability to distinguish between benign and malicious behavior
- Enhancement to resist against implicit-flows evasion
  - Treat all data written by an untrusted application to be tainted

  $\Rightarrow$ Fine-grained taint-tracking *does not* provide a benefit over a coarse-grained, conservative technique

Outline

Evasions

- Attacks by corrupting the shared address space
  - Without touching "sensitive" data
    - Corrupt a file descriptor rather than data that is written
    - Corrupt domain name (rather than cookies) within a data structure that keeps track of associations between them
- Attacking *attribution* mechanisms
  - Modify browser data so that it executes code paths chosen by BHO
  - Violate stack conventions, e.g., return-to-libc attack
  - Violate ABI conventions
- Attacking meta-data integrity
  - A BHO $\mathcal{M}$ races with a benign BHO or core browser to operate on sensitive data having them marked as *untainted*

Outline

Evasion
Known Evasions

- The underlying problems faced by the analysis are undecidable in general (as noted by the authors)
  - A condition $\mathcal{C}$ based on one-way hash functions
  - Exploration of unbounded number of branches
- However, attacks that exploit these problems may trigger suspicion and prompt a more detailed analysis by an expert.

Our goal: develop attacks that are unlikely to raise suspicion

Evasion

Known Evasions

- The underlying problems faced by the analysis are undecidable in general (as noted by the authors)
  - A condition $\mathcal{C}$ based on one-way hash functions
  - Exploration of unbounded number of branches
- However, attacks that exploit these problems may trigger suspicion and prompt a more detailed analysis by an expert.

Our goal: develop attacks that are unlikely to raise suspicion

Evasion

Using Memory Errors for Evasion

```
int trigger;
void procInput(void) {
   int *p = &buf[0];
   char buf[4096];
   ...
   gets(buf);
   ...
   *p = 1;
   ...
   if (trigger)
      malcode();
}
```

- `trigger` has to be marked *tainted*, to disclose `malcode`
- `trigger` is never tainted unless `p` points to it
- Deciding whether `p` could point to `trigger` is undecidable
- ... but the analysis proposed by Moser *et al.* could potentially *detect* the overflow of `buf` *is possible*
  - ⇒ not stealthy-enough ...

Idea: Make it difficult to discover the possibility of memory error, and to generate an input that exploits it.

Evasion
A Stealthier Technique

```
...
int trigger=0;
...
void procInput(void) {
   int pad, n, l;
   char buf[4096+256];
   int *p = &pad; char *dst;
   ...

    n = read(s, buf, sizeof (buf));
    l = computespace(buf, n);
    dst = alloca(l + 128);
    decode(buf, l, dst);

   ...
   *p = 1;
   ...
   if (trigger)
      malcode();
}
```

Evasion

Making Harder to Infer Functions' Properties

```
int computespace(char *src, int nread) {
  int i, k = 0;
  for (i = 0; i < nread; i++) {
    switch(src[i]) {
      case 0: k++; break;
      ...
      case 255: k++; break;
    }
  }
  return k;
}
```

- `computespace` is easy to compute
- . . . but it's hard to automatically understand that, at the end of the computation, `k` is equal to the length of `buf`

Evasion
Introducing the Vulnerable Condition

```
void decode(char *src, int nread, char *dst) {
  int i, j;
  for (i = 0, j = 0; i < nread; i++, j++) {
    switch(src[i]) {
      case 0: dst[j] = src[i]; break;
      ...
      case 113: dst[j++] = src[i];
                dst[j] = src[i];
                break;
      case 114: dst[j] = src[i]; break;
      ...
      case 255: dst[j] = src[i]; break;
    }
  }
}
```

- decode introduces the condition for an overflow to occur

$\Rightarrow$ dst overflows into p under certain conditions

  - The overflow *detection* requires 256^{127} tests on the average
  - Detection alone, however, *does not* disclose the malicious code

Conclusions

- Information flow techniques have been studied for decades
- Dynamic tainting techniques are quite robust in the context of software from trusted sources
- Promising results have been achieved by using these techniques for malware containment and analysis
  - However, malware writers can easily adapt their code to evade dynamic taint analysis

Utility of taint analysis is rather limited in the context of today's binary-based software deployment models

- Need to develop additional analysis techniques that complement information flow