What role for static analysis in malware detection?

Laurence Tratt
http://tratt.net/laurie/

Middlesex University

With thanks to David Clark

2011/4/6

Overview

1. What is malware and how do we traditionally detect it?
2. What is static analysis?
3. How does static analysis promise to help detect malware?
4. How far can we go with it?

What is malware?

- Malign software: infiltrates and subverts.
- Uses from spam e-mail botnets to IP theft.

What is malware?

- Malign software: infiltrates and subverts.
- Uses from spam e-mail botnets to IP theft.
- Executive summary: **malware is bad.**

How do we detect malware?

- Traditionally: signature (‘fingerprint’) detection.
- If a binary matches a malware signature, it’s a bad ’un.
- [Note: the signature may be for part(s) of a malware.]

How to defeat traditional signature matching.

- Original malware:

```
MOV R0, #3                          x := 3
BL  DO_SOMETHING_WITH_R0            f(x)
```

Give it hash *H*.

How to defeat traditional signature matching.

- Original malware:

```
MOV R0, #3                         x := 3
BL  DO_SOMETHING_WITH_R0           f(x)
```

  Give it hash H.

- Malware author (remember: bad, not mad) obfuscates it to:

```
MOV R0, #3                         x := 3
MOV R1, #4                         y := 4
BL  DO_SOMETHING_WITH_R0           f(x)
```

  Will have hash H' where $H \neq H'$.

How to defeat traditional signature matching (2).

- Idea: can signatures be like regular expressions, ‘skipping’ over irrelevant stuff?

How to defeat traditional signature matching (2).

- Idea: can signatures be like regular expressions, 'skipping' over irrelevant stuff?
- Original malware:

```
MOV R0, #3                          x := 3
BL  DO_SOMETHING_WITH_R0            f(x)
```

How to defeat traditional signature matching (2).

- Idea: can signatures be like regular expressions, ‘skipping’ over irrelevant stuff?
- Original malware:

```
MOV R0, #3                          x := 3
BL  DO_SOMETHING_WITH_R0            f(x)
```

- Malware author obfuscates it to:

```
MOV R0, #1                          x := 1
ADD R0, R0, #2                      x += 2
BL  DO_SOMETHING_WITH_R0            f(x)
```

How to defeat traditional signature matching (2).

- Idea: can signatures be like regular expressions, ‘skipping’ over irrelevant stuff?
- Original malware:

```
MOV R0, #3                         x := 3
BL  DO_SOMETHING_WITH_R0           f(x)
```

- Malware author obfuscates it to:

```
MOV R0, #1                         x := 1
ADD R0, R0, #2                     x += 2
BL  DO_SOMETHING_WITH_R0           f(x)
```

- No regular expression matching will match that!

How to defeat traditional signature matching (2).

- Idea: can signatures be like regular expressions, ‘skipping’ over irrelevant stuff?
- Original malware:

```
MOV R0, #3                          x := 3
BL  DO_SOMETHING_WITH_R0            f(x)
```

- Malware author obfuscates it to:

```
MOV R0, #1                          x := 1
ADD R0, R0, #2                      x += 2
BL  DO_SOMETHING_WITH_R0            f(x)
```

- No regular expression matching will match that!
- Metamorphic / polymorphic malware on the rise.
- Traditional signature detection ever less effective.

A proposed approach.

- Traditional signature detection looks at program syntax.

A proposed approach.

- Traditional signature detection looks at program syntax.
- What about the programs semantics?
- Intuition: a malware's core semantics must be the same before and after obfuscation.
- So:

A proposed approach.

- Traditional signature detection looks at program syntax.
- What about the programs semantics?
- Intuition: a malware's core semantics must be the same before and after obfuscation.
- So: we need to statically analyse its semantics!

Static analysis.

- Looking at a static program (source code or binary) and uncovering information about it.
- Take LLVM's static analyser (`scan-build`). Spot the bug?

```
char *expand_path(const char *path)
{
    char *exp_path;
    // If path begins with "~/", we expand that to the users home directory.
    if (strncmp(path, HOME_PFX, strlen(HOME_PFX)) == 0) {
        struct passwd *pw_ent = getpwuid(geteuid());
        if (pw_ent == NULL) {
            free(exp_path);
            return NULL;
        }

        if (asprintf(&exp_path, "%s%s%s", pw_ent->pw_dir, DIR_SEP, path +
          strlen(HOME_PFX)) == -1)
            errx(1, "expand_path: asprintf: unable to allocate memory");
    }
    else {
        if (asprintf(&exp_path, "%s", path) == -1)
            errx(1, "expand_path: asprintf: unable to allocate memory");
    }

    return exp_path;
}
```

Static analysis.

- Looking at a static program (source code or binary) and uncovering information about it.
- Take LLVM's static analyser (`scan-build`). Spot the bug?

```
char *expand_path(const char *path)
{
    char *exp_path;
    // If path begins with "~/", we expand that to the users home directory.
    if (strncmp(path, HOME_PFX, strlen(HOME_PFX)) == 0) {
        struct passwd *pw_ent = getpwuid(geteuid());
        if (pw_ent == NULL) {
            free(exp_path);
            return NULL;
        }

        if (asprintf(&exp_path, "%s%s%s", pw_ent->pw_dir, DIR_SEP, path +
          strlen(HOME_PFX)) == -1)
            errx(1, "expand_path: asprintf: unable to allocate memory");
    }
    else {
        if (asprintf(&exp_path, "%s", path) == -1)
            errx(1, "expand_path: asprintf: unable to allocate memory");
    }

    return exp_path;
}
```

Static analysis (2).

```
784
785             cur_ext->working = false;
786             if (conf->mode == DAEMON_MODE) {
787                 // If we're in daemon mode then, if this external has been found
788                 // not to be working, check the timeout (if it exists). If the
789                 // timeout hasn't been exceeded, then we have to give up on
790                 // trying to send this messages via this, or other, externals -
791                 // we need to wait for the timeout to be exceeded.
792                 if (cur_ext->timeout != 0 &&
793                   cur_ext->last_success + cur_ext->timeout > time(NULL)) {
794                     goto fail;
795                 }
796             }
797
798             cur_ext = cur_ext->next;
799         }
800     }
801
802 fail:
803     for (int j = 0; j < nargv; j += 1)
```

7 Loop condition is false. Execution continues on line 805

```
804         free(argv[j]);
805     free(argv);
806     free(stderr_buf);
807     free(dhd_buf);
```

8 Pass-by-value argument in function call is undefined

```
808
809     return false;
810 }
811
812
```

Static analysis (2).

```
    argv[i] = arg;
}
argv[nargv] = NULL;

// Setup a buffer into which we will read stderr from any child processes.

size_t stderr_buf_alloc = STDERR_BUF_ALLOC;
char *stderr_buf = malloc(stderr_buf_alloc);
if (stderr_buf == NULL) {

  4  Taking false branch

    syslog(LOG_CRIT, "try_groups: malloc: %m");
    exit(1);
}

// We now need to record where the actual message starts.

off_t mf_body_off = lseek(fd, 0, SEEK_CUR);
if (mf_body_off == -1) {

  5  Taking true branch

    syslog(LOG_ERR, "Error when ftell'ing from '%s'", msg_path);
    goto fail;

      6  Control jumps to line 803

}

// Read in the messages header, doctoring it along the way to make it
// suitable for being searched with regular expressions. The doctoring is
// very simple. Individual headers are often split over multiple lines: we
// merge such lines together.

size_t dhb_buf_alloc = HEADER_BUF;
```

Static analysis (3).

- Intuition: do a ‘fuzzy match’ against a malware’s *semantic signature* and that of a new binary.

Static analysis (3).

- Intuition: do a ‘fuzzy match’ against a malware’s *semantic signature* and that of a new binary.
- If they match: it’s a malware; otherwise it’s OK.
- (We might need to play around with the ‘fuzziness’ a bit, but it should work.)

Static analysis (3).

- Intuition: do a ‘fuzzy match’ against a malware’s *semantic signature* and that of a new binary.
- If they match: it’s a malware; otherwise it’s OK.
- (We might need to play around with the ‘fuzziness’ a bit, but it should work.)
- My argument: **if you deploy this tomorrow, by the following day it will have been irrevocably circumvented.**
- Why?

Static analysis assumptions.

- Underlying assumption of static analysis:

Static analysis assumptions.

- Underlying assumption of static analysis: programs are amenable to static analysis techniques and when a part of a program violates a static analysis technique, users are happy to adjust their program accordingly.

Static analysis assumptions.

- Underlying assumption of static analysis: programs are amenable to static analysis techniques and when a part of a program violates a static analysis technique, users are happy to adjust their program accordingly.

Bunnies and photo: Anna Hull. (CC BY-NC-ND 3.0)

The *pink fluffy bunny* assumption.

Static analysis assumptions (2).

- The pink fluffy bunny assumption breaks down with malware:

Static analysis assumptions (2).

- The pink fluffy bunny assumption breaks down with malware: malware authors will find and exploit any and all weak points.

Static analysis assumptions (2).

- The pink fluffy bunny assumption breaks down with malware: malware authors will find and exploit any and all weak points.

The *hostile* assumption.

Can we defeat the static analysis of malware?

- Consider a self encrypting malware.
- Consists of an initial decoder and an encrypted body.
- The following ARM(ish) code decrypts the data (w/length *lp*) and stores it back for execution.

```
   MOV R0, #0
   MOV R1, BODY
L: LDR R2, R1[R0]
   XOR R2, R2, #constant
   STR R2, R2[R0]
   ADD R0, R0, #4
   CMP R0, lp
   BLT L
BODY:
   encrypted malware body
```

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;


}
```

Can we defeat the static analysis of malware?

- Consider a self encrypting malware.
- Consists of an initial decoder and an encrypted body.
- The following ARM(ish) code decrypts the data (w/length *lp*) and stores it back for execution.

```
   MOV R0, #0
   MOV R1, BODY
L: LDR R2, R1[R0]
   XOR R2, R2, #constant
   STR R2, R2[R0]
   ADD R0, R0, #4
   CMP R0, lp
   BLT L
BODY:
   encrypted malware body
```

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;


}
```

Can we defeat the static analysis of malware?

- Consider a self encrypting malware.
- Consists of an initial decoder and an encrypted body.
- The following ARM(ish) code decrypts the data (w/length *lp*) and stores it back for execution.

```
   MOV R0, #0
   MOV R1, BODY
L: LDR R2, R1[R0]
   XOR R2, R2, #constant
   STR R2, R2[R0]
   ADD R0, R0, #4
   CMP R0, lp
   BLT L
BODY:
   encrypted malware body
```

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;


}
```

- What's its semantic signature?

Can we defeat the static analysis of malware (2)?

- First thought: the decrypter is basically an XOR in a loop...

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;
}
```

- ...and `body` points to a constant chunk of data.

Can we defeat the static analysis of malware (2)?

- First thought: the decrypter is basically an XOR in a loop...

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;
}
```

- ...and `body` points to a constant chunk of data.
- Should be quite easy to statically analyse and obtain a signature.

Can we defeat the static analysis of malware (3)?

- The decryption key is central.
- It must be a constant.
- Pink fluffy bunny assumption: the key must be transparently contained in the binary.

Can we defeat the static analysis of malware (3)?

- The decryption key is central.
- It must be a constant.
- Pink fluffy bunny assumption: the key must be transparently contained in the binary.

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;
}
```

Can we defeat the static analysis of malware (3)?

- The decryption key is central.
- It must be a constant.
- Pink fluffy bunny assumption: the key must be transparently contained in the binary.

```
int *body = ...;
for (int i = 0; i < lp; i += 1) {
    int t = body[i];
    t = t ^ constant;
    body[i] = t;
}
```

- Hostile assumption: the key can be opaquely calculated by the binary.

Hiding the key.

- Can we hide the key so that it can’t easily be uncovered?

Hiding the key.

- Can we hide the key so that it can't easily be uncovered?
- Let's make it a lot harder:

```
int k;
for (int i = 0; i < MAXINT; i += 1) {
    if (md5(i) == constant1 && sha256(i) == constant2) {
        k = i;
        break;
    }
}
```

- `constant1` and `constant2` are in the binary, but aren't directly related to `k`.
- To statically analyse that, we need to analyse the MD5 and SHA256 functions.

Hiding the key.

- Can we hide the key so that it can't easily be uncovered?
- Let's make it a lot harder:

```
int k;
for (int i = 0; i < MAXINT; i += 1) {
    if (md5(i) == constant1 && sha256(i) == constant2) {
        k = i;
        break;
    }
}
```

- `constant1` and `constant2` are in the binary, but aren't directly related to `k`.
- To statically analyse that, we need to analyse the MD5 and SHA256 functions.
- Hash functions are meant to be hard to analyse; but not without their weaknesses.

Hiding the key.

- Can we hide the key so that it can't easily be uncovered?
- Let's make it a lot harder:

```
int k;
for (int i = 0; i < MAXINT; i += 1) {
    if (md5(i) == constant1 && sha256(i) == constant2) {
        k = i;
        break;
    }
}
```

- `constant1` and `constant2` are in the binary, but aren't directly related to `k`.
- To statically analyse that, we need to analyse the MD5 and SHA256 functions.
- Hash functions are meant to be hard to analyse; but not without their weaknesses.
- Take the hostile assumption: make it harder!

Hiding the key (2).

- Try statically analyzing random data:

```
int k;
f = open("/dev/random", "r");
while (true) {
    int t = readc(f) | (readc(f)«8) | (readc(f)«16) | (readc(f)«24);
    if (md5(t) == constant1 && sha256(t) == constant2) {
        k = t;
        break;
    }
}
```

- Rough speed: in C, will find a key corresponding to the hash of a 5 character string on my laptop in under a minute.

Hiding the key (2).

- Try statically analyzing random data:

```
int k;
f = open("/dev/random", "r");
while (true) {
    int t = readc(f) | (readc(f)«8) | (readc(f)«16) | (readc(f)«24);
    if (md5(t) == constant1 && sha256(t) == constant2) {
        k = t;
        break;
    }
}
```

- Rough speed: in C, will find a key corresponding to the hash of a 5 character string on my laptop in under a minute.
- Moser, Kreugel, and Kirda show examples of opaque constants whose static solution would be equivalent to solving an NP-hard problem.

Can limited dynamic analysis help?

- Opaque constants defeat static analysis on its own.
- Can we dynamically run the malware decrypter, stop it, and then semantically analyse the decrypted malware?

Can limited dynamic analysis help?

- Opaque constants defeat static analysis on its own.
- Can we dynamically run the malware decrypter, stop it, and then semantically analyse the decrypted malware?
- Take the hostile assumption: will embed more than one layer of hard to analyse encryption.

What are the limits of static analysis?

- Assertion: static analysis of malware on its own would quickly be circumvented (by the hostile assumption).
- Could static analysis have any use in malware detection?

What are the limits of static analysis?

- Assertion: static analysis of malware on its own would quickly be circumvented (by the hostile assumption).
- Could static analysis have any use in malware detection? Yes!

What are the limits of static analysis?

- Assertion: static analysis of malware on its own would quickly be circumvented (by the hostile assumption).
- Could static analysis have any use in malware detection? Yes!
    1. In security labs analyzing malware (every tool helps).

What are the limits of static analysis?

- Assertion: static analysis of malware on its own would quickly be circumvented (by the hostile assumption).
- Could static analysis have any use in malware detection? Yes!
  1. In security labs analyzing malware (every tool helps).
  2. In an interleaved dynamic / static analysis.

Further reading

- *Static Analysis for Malware Detection* Andreas Moser, Christopher Kruegel, Engin Kirda.

Summary

- Static analysis of malware has assumed a pink fluffy bunny world.
- In a hostile world, everything changes: malware authors will create self-encrypted malware using opaque constants.

Summary

- Static analysis of malware has assumed a pink fluffy bunny world.
- In a hostile world, everything changes: malware authors will create self-encrypted malware using opaque constants.
- *But* there are uses for it, but not the ones that there first appeared to be.

Summary

- Static analysis of malware has assumed a pink fluffy bunny world.
- In a hostile world, everything changes: malware authors will create self-encrypted malware using opaque constants.
- *But* there are uses for it, but not the ones that there first appeared to be.
- A general rule: anything that relies on static analysis for security must bear in mind the hostile assumption at all times.

Summary

- Static analysis of malware has assumed a pink fluffy bunny world.
- In a hostile world, everything changes: malware authors will create self-encrypted malware using opaque constants.
- *But* there are uses for it, but not the ones that there first appeared to be.
- A general rule: anything that relies on static analysis for security must bear in mind the hostile assumption at all times.

Thanks for listening