Auditing Open Source Perl Code for Security

John Lightsey

Why do it?

- Fun
- Learning experience
- Community

Review vs Audit

- Review implies code is unreleased
- Audit implies code is released

Formal Approach

- Prep
- Model
- Audit
- Quantify

Formal: Prep

- Understand the business logic
- Understand the execution context

Formal: Model

- Generate a threat model
  - Define threat agents
  - Define attack surfaces
  - Define possible attacks
  - Define required security controls
  - Define potential impacts

Formal: Audit

- Review the code to determine the effectiveness of the security controls

Formal: Quantify

- Classify discovered defects
- Generate metrics
- Refine approach

Formal Approach

- Lots of prep work
- Deep understanding of the application

Code Crawling

- Scan code for key pointers to vulnerabilities
  - Vulnerable source
  - Vulnerable sink
  - Lack of security controls between the two

Code Crawling: Advantages

- Simple to initiate
- Vulnerabilities are easy to demonstrate and understand
- Lots of documentation on these types of vulnerabilities

Code Crawling: Disadvantages

- Responsibility for security controls
- Overall logic flaws
- Not a good basis to build a better foundation

Where to start

- Perl blead branch
- http://grep.cpan.me
- http://szabgab.com/perl-based-open-source-products.html
- http://debtags.debian.net
- http://codesearch.debian.net
- http://sourceforge.net
- http://github.com

Code Crawling: Approach

- Start with sources
- Start with sinks
- Start with source + sink combination

Code Crawling: Work Flow

- Identify suspect sinks
- Follow code execution back to vulnerable source
- Verify no effective security controls between the two
- +1 Create POC
- +1 Create patch
- +1 Document well
- Disclosure

Don't start what you can't finish!

- Disclosure is very time consuming
- Guidelines on how disclosure works are very poor
- Organizations are very inconsistent
- Keep notes on every step of the process

Disclosure: Passive-Aggressive Approach

- Hint but don't say

Disclosure: Black Hat Approach

- Keep it secret and use for $$profit$$
  - Free medical and dental coverage
  - Free gym access
  - Free meals

Disclosure: Ambush Approach

- Disclose fully to a public channel without contacting the upstream authors privately first
  - Quick
  - Easy
  - Forces a rapid response
  - Not friendly

Disclosure: Responsible Approach

- Initial contact with upstream
  - Verify secure communications first
  - Provide all documentation
  - +1 Specify timelines for response
  - +1 Specify action if no response
  - +1 Request credit

Disclosure: Responsible Approach

- Continued Communication
  - Provide further details in response to the upstream author's questions
  - Update the timelines
  - +1 Upstream author requests CVE number from cve-assign@mitre.org

Disclosure: Responsible Approach

- Patch development
  - Help with the proposed fix
  - Upstream may view you as an expert

Disclosure: Responsible Approach

- Public disclosure by upstream author
  - No consistency here
  - +1 Request CVE via oss-sec list if upstream author did not do so

Disclosure: Responsible Approach

- Proof of Concept disclosure
  - Step is optional
  - May result in unpatched systems being attacked
  - May result in similar vulnerabilities being found by others

Disclosure: Responsible Approach

- Likely pitfalls
  - No response to initial contact
  - No response after details are provided
  - Request for unreasonable delays
  - Fixed without acknowledgement of security nature
  - Fix documented in obscure location
  - Author believes their code is not responsible
  - No credit given

Don't start what you can't finish!

- Once you discover a vulnerability it's difficult to keep it secret

Code Crawling: My Work Flow

- Get the latest release
- Check in to git
- Run macros for suspect sinks
- Trace execution backwards
- POC, Patch, Disclosure, etc

Sinks: Duck

- $obj->can($input)
- $package->can($input)
- $obj->$method()
- Some::Package->$method

```
can="!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"\\->\\s*can\\s*\\(\\" -- $GIT_PREFIX'"

duck="!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"\\->\\s*\\$\\" -- $GIT_PREFIX'"
```

Duck: Attacks

- Code: $method = $param;
- Attack: $param = “Some::Other::function”;
- Attack: $param = “_private_method”;

- Code: $method = $param . '_safe';
- Attack: $param = “Some::Other::function\0”;

Duck: Example

```
# Safe->reval()  (before version 2.35)
my $injection = <<'EO_INJECTION';
my $obj =  bless {
    Mask => ( "\x{00}" x 47), # opcode mask size
    Root => "DoesntMatterButMustBeSet",
}, __PACKAGE__;
sub wrap_code_refs_within {}
$SIG{CHLD} = sub {
    my $method = "Safe::reval";
    $obj->$method("eval q( print qq|Injection as | . `whoami`);" );
};
return 1;
EO_INJECTION
```

Duck: Example

```
# Locale::Maketext before 1.23 CVE-2012-6329
elsif($m =~ /^\w+(?:\:\:\w+)*$/s
      and $m !~ m/(?:^|\:)\d/s
      # exclude starting a (sub)package or symbol with a digit
   ) {
    # Yes, it even supports the demented (and undocumented?)
    #  $obj->Foo::bar(...) syntax.
```

Attack: $lh->maketext(“[Some::Other::function,arg2,arg3]”);

Sinks: Eval

- eval() with variable interpolation
- Safe::reval
- Safe::rdo

```
eval="!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"(^|::|\\s+|\\(\\s*)(r?eval|rdo)\\s*\\(?\\s*(q|\")'\\" --
$GIT_PREFIX"'
```

Eval: Example

```
# Locale::Maketext before 1.23 CVE-2012-6329
my $sub = eval(join '', @code);
```

Attack: $lh->maketext('[quant,1\,);`whoami`;#]');

Sinks: Exec

- Any form of exec with variable interpolation
- TODO: Add open()

```
exec="!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"(^|\\s+|\\(\\*)(`|(exec|system|qx)(\\s*\\
(|\\s+\"|\\s+'\\"\\'\\"\\"|\\s+qq?))'\\" --
$GIT_PREFIX'"
```

Sinks: IDs

- Any UID or GID change
  - Saved UID/GID
  - Environment leakage
  - RLIMIT_NPROC
  - Supplemental groups

```
sec-ids-all = "!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"(^|\\s+|\\(\\s*)\\$(<|>|\\(|\\))\\s*=[^=]'\\" --
$GIT_PREFIX'"
```

IDs: Example

FCGI::Spawn

```
my $gid = getgrnam( $group );
die "Get group $group: $!" if $gid == 0;
setgid( $gid );
$) = $gid;
$( = $gid;
die "Set group $group($gid): $!" if ( $( != $gid ) or ( $) != $gid );
```

IDs: Example

Not currently fixed

Sinks: Temp

- Anything that touches a tmp directory
  - Predictable locations
  - Dependent locations
  - Insecure permissions

```
temp="!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"(/tmp/|TE?MP|mktemp|tmpdir)'\\" --
$GIT_PREFIX"'
```

Temp: Example

PAR::Packer before 1.012 CVE-2011-4114

> par_mktmpdir() makes no effort to verify that the
> /tmp/par-<username> directory is safe to use (owned by the correct
> UID and GID, not world writable, no symlinks in the path that are
> owned by another user.)
>
> This makes PAR packed scripts unsafe on multiuser systems.

Yawn. Where does it say that they are safe?

If you're really concerned about safety you should use per-user temp directories, not for PAR::Packer, but in general.

Cheers, Roderich

Temp: Example

Parallel::ForkManager before 1.0.0
CVE-2011-4115

Sinks: Thaw

- Storable
- YAML
- XML
- Others?

```
thaw="!/bin/bash -c 'git grep -n -A 3 -E -e '\\"(^|::|\\s+|\\
(\\s*)(thaw|(lock_|fd_|p)?
retrieve|Load(File)?|XMLin|parsefile)(\\s*\\(|\\s+)'\\" --
$GIT_PREFIX"'
```

Thaw: Example

Storable::thaw() on cookie data for session persistence:

App::Context (CVE-2013-6141)

HTML::Ep (CVE-2013-6142)

Spoon (CVE-2013-6143)

Thaw: Example

Not currently fixed

Thaw: Example

Not currently fixed

Sinks: I18N

- s?printf
- maketext
- Any other localization support

```
i18n="!/bin/bash -c 'git grep -n -A 3 -E -e
'\\"(^|->|\\s+|\\(\\s*)(maketext|s?printf)(\\s*|\\s*\\
(\\s*)(\"[^\"]*\\$|qq|\\$)'\\" -- $GIT_PREFIX'"
```

I18N: Example

YaBB current version (patch available) CVE-2013-2057

```
if ($yyCookies{'guestlanguage'} && !$FORM{'guestlang'} &&
$enable_guestlanguage) {
    $language = $guestLang = $yyCookies{'guestlanguage'};
}
...
if (-e "$helpfile/$language/$help_area/$_.help") {
    require "$helpfile/$language/$help_area/$_.help";
}
```

Attack: guestlanguage=../../../public_html/yabbfiles/Attachments/test.txt%00

I18N: Example

FosWiki CVE-2013-1666

TWiki CVE-2013-1751

```
 # escape everything:
 $str =~ s/\[/~[/g;
 $str =~ s/\]/~]/g;

 # restore already escaped stuff:
 $str =~ s/~~\[/~[/g;
 $str =~ s/~~\]/~]/g;
```

%MAKETEXT{"~~[quant,4, singular, plural, ~~]"}%

Sinks: TODO

- SQL
- Open
- XS
- ???

The Point

- Have fun
- Learn something
- Help others

Links

- https://www.owasp.org/index.php/OWASP_Code_Review_Guide_Table_of_Contents
- https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents
- http://perltraining.com.au/notes/perlsec.pdf
- https://www.securecoding.cert.org/confluence/display/perl/CERT+Perl+Secure+Coding+Standard