Off BEAM | ɒf biːm |

informal on the wrong track; mistaken:
“You're way off beam on this one”

Bram Verburg

https://blog.voltone.net/
@voltonez

KUNGL TEKNISKA HÖGSKOLAN

SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Making reliable distributed systems in the presence of software errors

Final version (with corrections) — last update 20 November 2003

Joe Armstrong

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfilment of the requirements for
the degree of Doctor of Technology
The Royal Institute of Technology
Stockholm, Sweden

“Secure Coding”

- Part of a *Secure Software Development Life Cycle*
- One of many activities
- Main strength: catch things early
- Programmers’ “muscle memory”

Do not do this, do that!

- C: Do not use `strcpy()`, use `strncpy()`
- JavaScript (DOM): Do not set `.innerHTML`, set `.innerText`
- BEAM: Do not use `list_to_atom/1`, use `list_to_existing_atom/1`

Erlang Ecosystem Foundation

Security Working Group

https://erlef.github.io/security-wg/

Contents

Preventing atom exhaustion

- Not just the `to_atom/1` functions
- Interpolation in Elixir:
  - `"new_atom_#{index}"`
  - `~w[row_#{index} column_#{index}]a`
- Library functions:
  - Erlang standard library: xmerl
  - 3rd party packages

Serialisation and deserialisation

- *External Term Format (ETF)* is not for External use!
- Stick to JSON, XML, Protobuf, TOML, etc. for interactions:
  - With untrusted elements, or
  - Over an untrusted channel
- `term_to_binary/2` ‘safe’ mode is not safe:
  - Unsafe data types, including anonymous functions

```elixir
themes =
  case conn.cookies["themes"] do
    nil ->
      []

    themes_b64 ->
      themes_b64
      |> Base.decode64!()
      |> :erlang.binary_to_term([:safe])
  end

css = Enum.map(themes, &theme_to_css/1)
```

Deserialisation

Elixir, using `:erlang.binary_to_term/2`

```elixir
# Attacker generates:
pwn = fn _, _ -> IO.puts("Boom!"); {:cont, []} end
cookie =
  pwn
  |> :erlang.term_to_binary()
  |> Base.encode64()

# Server executes:
Enum.map(pwn, &theme_to_css/1)

# Exercise for the reader: what would happen with this input:
cookie =
  1..9999999999999999
  |> :erlang.term_to_binary()
  |> Base.encode64()
```

Deserialisation

Elixir, using `:erlang.binary_to_term/2`

Serialisation and deserialisation

- `Plug.Crypto.non_executable_binary_to_term/1,2`
  - Prevents deserialisation of functions
  - Remember to pass `:safe` as well
- Most of the time, don't use ETF
  - Make sure the parser is atom-safe

Spawning external executables

- Do not use:
  - `os:cmd/1,2`
  - `open_port/2` with `{spawn, Command}`
- Instead, use:
  - `open_port/2` with `{spawn_executable, FileName}` and `args`
- Do not use a shell with `spawn_executable`

```erlang
lolcat(Text) ->
  Command =
    "convert lolcat.jpg -gravity south "
      "-stroke '#000C' -strokewidth 2 -pointsize 36 "
      "-annotate 0 \"" ++ Text ++ "\" "
      "-stroke  none   -fill white    -pointsize 36 "
      "-annotate 0 \"" ++ Text ++ "\" "
      "result.jpg",
  os:cmd(Command).


% User enters "$DB_PASSWORD"
```

Spawning external executables

Erlang, using `os:cmd/1`

Spawning external executables

Erlang, using `os:cmd/1`

```erlang
lolcat(Text) ->
  Command =
    "convert lolcat.jpg -gravity south "
      "-stroke '#000C' -strokewidth 2 -pointsize 36 "
      "-annotate 0 \"" ++ Text ++ "\" "
      "-stroke  none   -fill white    -pointsize 36 "
      "-annotate 0 \"" ++ Text ++ "\" "
      "result.jpg",
  os:cmd(Command).


% User enters "$DB_PASSWORD"


% User enters "$(which cat)"
```

Spawning external executables

Erlang, using `os:cmd/1`

Spawning external executables

Erlang, using `os:cmd/1`

Spawning external executables

- Elixir standard library: `System.cmd/2,3` uses `open_port`:
  - With `spawn_executable` and `args`
  - Locates executable in $PATH
  - Wrapper to return output
  - Do not use a shell as the command!
- Beware of inherited environment with sensitive data:
  - Remove variables with `env` argument to `open_port/2` / `System.cmd/3`

Protecting sensitive data

- Immutable data structures
- Garbage collection
- Logging and exceptions
- Crash dumps
- Introspection

Protecting sensitive data

- Passing closures
- Purging stack traces
- Private ETS tables
- Implement `format_status/2` callback
  - For `gen_server`, `gen_event` or `gen_statem`

```
1> WrappedKey = fun() -> "SuperSecretKey" end.
#Fun<erl_eval.20.128620087>

2> crypto:mac(hmac, sha256, "Message", WrappedKey()).
<<129,105,141,237,112,6,98,183,249,80,221,2,209,84,117,
  185,148,11,173,45,66,236,187,150,74,36,43,244,19,...>>

3> crypto:mac(hmac, sha256, undefined, WrappedKey()).
** exception error: {badarg,{"mac.c",216},"Bad key"}
     in function  crypto:mac_nif/4
        called as crypto:mac_nif(hmac,blake2,undefined,"SuperSecretKey")
```

Exception, leaking HMAC key

Erlang, unwrapping key to pass to `crypto`

```
mac(Type, Digest, Message, WrappedKey) ->
    try
        crypto:mac(Type, Digest, Message, WrappedKey())
    catch
        Class:Reason:Stacktrace0 ->
            Stacktrace = prune_stacktrace(Stacktrace0),
            erlang:raise(Class, Reason, Stacktrace)
    end.

prune_stacktrace([{M, F, [_ | _] = A, Info} | Rest]) ->
    [{M, F, length(A), Info} | Rest];

prune_stacktrace(Stacktrace) ->
    Stacktrace.
```

Stacktrace pruning

Erlang, unwrapping key to pass to `crypto`

```
1> WrappedKey = fun() -> "SuperSecretKey" end.
#Fun<erl_eval.20.128620087>

2> pruned:mac(hmac, sha256, "Message", WrappedKey).
<<129,105,141,237,112,6,98,183,249,80,221,2,209,84,117,
  185,148,11,173,45,66,236,187,150,74,36,43,244,19,...>>

3> pruned:mac(hmac, sha256, undefined, WrappedKey).
** exception error: {badarg,{"mac.c",216},"Bad key"}
     in function  crypto:mac_nif/4
     in call from pruned:hmac/3 (pruned.erl, line 12)
```

Stacktrace pruning

Erlang, unwrapping key to pass to `crypto`

#CodeBEAMSF

Protecting sensitive data

- In crypto libraries, combine the two:
  - Allow caller to pass in a closure with secret/key
  - Prune stack trace in function that unwraps the closure
- Plug/Phoenix applications:
  - Use `Plug.Crypto.prune_args_from_stacktrace/1`

Erlang standard library: ssl

- Client side:
  - `{verify, verify_peer}`, even with many libraries (HTTPS, ...)
  - Remember, OS CA trust store is an option
- Please watch my ElixirConf EU 2019 talk:
  - *“Learn you some :ssl for much security”*

Erlang standard library: xmerl

- `xmerl_scan` creates atoms:
  - For tag and attribute names
  - Note: popular Hex packages build on `xmerl_scan`
- `xmerl_sax_parser` vulnerable to “billion laughs” attack:
  - Raise an exception on `internalEntityDecl` and `externalEntityDecl` events

Boolean coercion in Elixir

- Elixir: anything other than `nil` or `false` is ‘truthy’
    - Erlang: no such thing as ‘truthy’, no such thing as `nil`, occasional `undefined`
- Used in:
    - Branching: `if`, `unless` and `cond`
    - Boolean algebra: `&&`, `||` and `!`

```
# verify/3 returns :ok | {:error, atom()}
:signature.verify(signature, message, private_key) ||
  raise(BadSignatureException)

# Use strict boolean 'or':
:signature.verify(signature, message, private_key) or
  raise(BadSignatureException)

# Or use ‘case', to be more explicit:
case :signature.verify(signature, message, private_key) do
  true ->
    # Do something

  false ->
    raise(BadSignatureException)
end
```

Boolean coercion

Elixir

Deployment hardening

- Installing/building the runtime system
- Releases
- Distribution
- Crash dumps and core dumps
- …

Tools and resources

- Static analysis:
  - Dialyzer, Credo, Sobelow
- Documentation:
  - OWASP
  - CIS Benchmarks

Erlang Ecosystem Foundation

Security Working Group

https://erlef.github.io/security-wg/

bluecode

Thank you!

Bram Verburg

https://blog.voltone.net/
@voltonez