Secure Coding in Elixir

por Matheus Anzzulin

Referência:

Documentação, especificações e cógido de Security Working Group of the Erlang Ecosystem Foundation
https://erlef.github.io/security-wg/secure_coding_and_deployment_hardening/introduction

Prevenir exaustão de átomos

- Use String.to_existing_atom/1 ao invés de String.to_atom/1
- Use List.to_existing_atom/1 ao invés de List.to_atom/1
- Use Module.safe_concat/1,2 ao invés de Module.concat/1,2
- Não use interpolações para criar átomos

```
:"new_atom_#{index}"
:'new_atom_#{index}'
~w[row_#{index} column_#{index}]a
```

- Use a opção :safe ao chamar :erlang.binary_to_term/2 em inputs não confiáveis (veja também Serialização e Desserialiazação)

Serialização e Desserialiazação

- Ao escolher uma serialização para ser utilizada em um ambiente não confiável, usea forma padrão (por exemplo: JSON, XML, Protobuf) ao invés de External Term Format (ETF)
  Use um parser que não gere átomos (veja também Prevenir Exaustão de átomos)

Gerando executáveis externos

Previna desserialização de funções de inputs não confiáveis, por exemplo usar Plug.Crypto.non_executable_binary_to_term/1,2

Exemplo

Considere o código em Elixir a seguir, de uma aplicação web hipotética que armazena temas customizáveis de UI em um cookie, usando ETF:

```
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)
```

Um usuário malicioso pode manipular o cookie:

```
# 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)
```

A função anonima do atacante seria executado no servidor, tornando isso uma vulnerabilidade de RCE.

Gerando executáveis externos

Ao usar System.cmd/2,3:
Evite usar uma sell com um executável e passar os argumentos como um única string
Limpe ou sobrescreva varias de ambiente sensíveis usando a opção :env

Na maioria das plataformas, o processo gerado irá herdar o ambiente do processo BEAM, o qual pode conter informações sensíveis. Por exemplo, ao utilizar variáveis de ambiente para passar credenciais do banco de dados para aplicação BEAM, garanta que a variável seja limpa na chamada para open_port/2 ou System.cmd/2,3, para diminuir o risco da senha vazar através de vulnerabilidades, memory dump ou comportamento imprevisível do sistema:

```
System.cmd("env", [], env: %{"DB_PASSWORD" => nil})
```

Protegendo dados sensíveis

- Implemente ou derive o protocolo de Inspect para estruturas
- Implemente o retorno format_status/2 para os processos GenServer, :gen_event ou :gen_statem com informações sensíveis
- Use a opção :private para tabelas ETS contendo informações sensíveis
- Sinaliza o processo atual como sensível utilizando :erlang.process_flag(:sensitive, true) para processos contendo dados sensíveis ou aplicações lógicas

Protegendo dados sensíveis

Há diversas formas que dados sensíveis, como senhas e chaves privadas, podem vazar:

- Um stack trace, mostrados no console ou os logs seguindo um exceção
- Mensagens de logs geradas por frameworks/bibliotecas
- Funções de instrospecção utilazadas para debugar ou monitorar, por exemplo utilizar os modulos erlang, sys ou dbg ou a ferramenta observadora
- Um memory dump, gerado quando o VM encontra um problema do qual não consegue se recuperar
- Um memory dump do OS, como resultado de uma falha interna nos executáveis BEAM ou outro código nativo

Protegendo dados sensíveis

Wrapping

Para prevenir informações sensíveis de vazarem em um stack trace, o valor pode ser envolvido em uma encerramento: uma função anonima de zero-arity. O valor interno pode facilmente ser unwraped onde é necessário invocando a função. Se um erro ocorre e os argumentos da função são mostrados em um console ou log, é mostrado como #Fun<...> ou #Function<...> . Segredos envolvidos em um encerramento também estão salvos de instrospecção utilizando o Observer e de serem escritos em memory dumps

```
wrapped_secret = fn -> System.get_env("SECRET") end
```

Protegendo dados sensíveis

Poda do stacktrace

Envolver a(s) chamada(s) da(s) função ou funções em uma expressão (Erlang) try … catch (Erlang) ou adicionar uma clausula de resgate ao corpo de uma função (Elixir), e despir os argumentos da função antes da exceção

```
def encrypt_with_secret(message, wrapped_secret) do
 ComeCryptoLib.encrypt(message, wrapped_secret.())
rescue
 e -> reraise e, prune_stacktrace(System.stacktrace())
end

defp prune_stacktrace([{mod, fun, [_ | _] = args, info} | rest]),
 do: [{mod, fun, length(args), info} | rest]

defp prune_stacktrace(stacktrace), do: stacktrace
```

(Adaptado do pacote plug_crypto, a função Plug.Crypto.prune_args_from_stacktrace/1 pode ser utilizada diretamente nas clausulas de resgate, se o pacote estiver disponivel)

Protegendo dados sensiveis

Customizando a instrospecção

No Elixir, quando termos precisam ser escritos no console ou em logs, o inspect é utilizado para gerar uma string representatido do termo. Ao customizar a implementação do protocolo inspect para estruturas é possível filtrar ou mascarar campos sensíveis. É também possível utilizar uma anotação @derive antes da definição de uma estrutura, selecionando so campos que devem ser inclusos ou não quando a extrutura é inspecionada
Para GenServer, :gen_event ou :gen_statem processes, implementar o chamada de controles the format_status/2 como o estado interno é representado por ferramentas de introspecção, como o 'observador'. Se o estado é um map, por exemplo, a função pode mascarar o valor para certas chaves

--

Protegendo dados sensiveis

Processos

Por fim, um processo pode ser marcado como 'sensível', usando erlang:process_flag/2. Tem os seguintes efeitos:

- O conteúdo de filas de mensagens não pode ser introspectado, e não é escrito para um crash dump
- Processos de dicionário não podem ser instrospectados, e não são escritos para uma crash dump
- O estado dos processos gen_server, gen_event ou gen_statem process não podem ser instrospectados ou escritos para um crash dump
- O heap e o stack do processo não são escritos para uma crash sump

Sandboxing códigos não confiiveis

- Não utilise Code.eval_file/1,2, Code.eval_string/1,2,3 ou Code.eval_quoted/1,2,3 em entradas não confiaveis ou em codigos em produção de jeito nenhum
- Use an embedded language runtime, such as Lua

Prevenindo ataques de oportunidade

- Use crypto:hash_equals/2 ou funções de comparação de constate de tempo similares ao invés de pattern matching ou operadores nativos para comparar segredos

Prevenindo ataques de oportunidade

Qual a diferença dessas 2 implementações?

```
case conn.assigns[:token] do
  ^token -> :ok
  _ -> :access_denied
end
```

```
case Plug.Crypto.secure_compare(conn.assigns[:token], token) do
  true -> :ok
  false -> :access_denied
end
```

Prevenindo ataques de oportunidade

A segunda implementação utiliza crypto:hash_equals/2. A checagem evita atalhos de comparação que tornariam vulneravél a um ataque de oportunidade. Note que é necessário que os dois argumentos tenham o mesmo tamanho, normalmmente o retorno de uma função hash.
O pacote plug_crypto de Elixir (o qual faz parte de qualquer aplicação Phoenix por padrão) contem secure_compare/2.

Coerção boleana em Elixir

- Prefira case ao inves de if, unless ou cond
- Prefira and ao invés de &&
- Prefira or ao invés de ||
- Prefira not ao invés de !