

serve-event-tricky.lisp

Luke Gorrie

May 15, 2004

Contents

1	Introduction	1
2	The server	1
3	The client	3
4	Test cases	3
4.1	Delay exploit	3
4.2	Error propagation exploit	4
5	Conclusion	5

1 Introduction

This is a small example to show some tricky aspects of using the `SERVE-EVENT` framework in CMUCL. First we define a simple server program, then we write some small clients to provoke it into behaving badly.

```
(in-package :cl-user)
```

```
(defvar port 10000
  "The TCP port for the server.")
```

We will often want to print short messages, so here is a helpful little utility:

```
(defun say (format &rest args)
  "Print a formatted message to standard-output on a fresh line."
  (format t "~&??%" format args)
  (force-output))
```

2 The server

The server binds a listening socket and loops accepting connections. For each connection it calls `READ`, prints the result, and closes the socket. All I/O scheduling is driven by `SERVE-EVENT`.

The server reports abnormal exits and unhandled conditions from handlers.

```

(defun run-server ()
  "Run the server in a loop until aborted."
  (let ((listen-socket (start-server)))
    (unwind-protect (server-loop)
      (stop-server listen-socket))))

(defun start-server ()
  "Start the server."
  (let ((socket (ext:create-inet-listener port :stream :reuse-address t))
        (nr-connections 0))
    (sys:add-fd-handler socket :input
      (lambda (s)
        (server-accept (incf nr-connections) s)))
    socket))

(defun server-loop ()
  "Loop serving requests until aborted.
If an error is raised then it is printed and the loop continues."
  (with-simple-restart (stop-server "Stop the example server.")
    (handler-case (loop (sys:serve-all-events))
      (error (err)
        (say "Continuing after error: ~A" (type-of err))))))

(defun stop-server (socket)
  (sys:invalidate-descriptor socket)
  (close-socket socket))

(defun server-accept (nr listen-socket)
  "Accept new connection number NR from LISTEN-SOCKET.
This is a callback for when SERVE-EVENT detects a new connection."
  (sys:add-fd-handler (accept-tcp-connection listen-socket)
    :input
    (lambda (socket)
      (server-handle-connection nr socket))))

(defun server-handle-connection (number socket)
  "Handle connection NUMBER on SOCKET.
Try to READ one sexp from the socket and print it.
Also print a message if we lose control (the stack unwinds) unexpectedly.
This is a callback for when SERVE-EVENT detects available data."
  (let ((stream (sys:make-fd-stream socket :input t))
        (successful nil))
    (unwind-protect
      (with-standard-io-syntax
        (say "Connection #~D read ~A" number (read stream))
        (setq successful t)))
      (close-connection stream)
      (unless successful (say "Connection #~D aborted!" number))))))

```

```
(defun close-connection (stream)
  (sys:invalidate-descriptor (sys:fd-stream-fd stream))
  (when (open-stream-p stream)
    (close stream)))
```

3 The client

The client part is just a few utilities for opening sockets and sending strings.

```
(defun make-client ()
  "Connect to the server and return the (unbuffered) output stream."
  (let ((fd (connect-to-inet-socket "localhost" port)))
    (sys:make-fd-stream fd :output t :buffering :none)))
```

```
(defmacro with-clients ((&rest variables) &body body)
  "Bind VARIABLES to new connections to the server and execute BODY.
  Ensure the connections are closed before returning."
  `(let ,(mapcar (lambda (var) (list var '(make-client))) variables)
    (unwind-protect
      (progn ,@body)
      (dolist (client (list ,@variables))
        (close-connection client))))))
```

```
(defun send (client string)
  "Send STRING to CLIENT and pause a moment to let the server process it."
  (princ string client)
  (sleep 0.1))
```

4 Test cases

Our task now is to write clients that provoke undesirable behaviour from the server. To test the programs you should start two Lisp systems and load this file in both. Then in one you do (`run-server`) and in the other you can call the exploit functions defined below.

An important tool is being able to make the server enter calls to `READ` and choosing when to allow them to exit. We can do this by sending a `SEXP` in two parts – the first causes the server to start reading, and it can't finish until we send the rest.

```
(defparameter first-half "(foo"
  "The first half of a SEXP.")

(defparameter second-half "bar)"
  "The other half of the SEXP.")
```

4.1 Delay exploit

Clients can case delays for each other. Here is a case with two clients, A and B, where A must wait for B to finish a request before being able to proceed, even though all of A's data is available to the server.

```
(defun delay ()
  (with-clients (a b)
    (send a first-half)
    (send b first-half)
    (send a second-half)
    (sleep 5)
    (send b second-half)))
```

The expected output is for the server to say nothing for a few seconds, and then print:

```
;; Connection #2 read (FOOBAR)
;; Connection #1 read (FOOBAR)
```

Here's why:

Both clients connect and send half of a request, with A sending first. The server enters "blocking" READs, first for A and then for B, and awaits more input. The relevant parts of the server's Lisp stack look like this:

```
;; (SERVE-EVENT)
;; (READ B)
;; (SERVE-EVENT)
;; (READ A)
;; (SERVE-EVENT)
;; (SERVER-LOOP)
```

Next A sends the rest of his request. But what can we do with it? Nothing yet: we cannot return from (READ A) without first returning from (READ B), and B is still blocking. A must wait for B's request to complete. In the test case this takes a few seconds.

This case can occur if the network fails between client and server, and it's easy to trigger deliberately (maliciously). A timeout of some kind is required to break out of the problem.

4.2 Error propagation exploit

Another issue presents itself from looking at the previous stack diagram. What if an unhandled condition is signalled in the READ of B? With our server it will propagate right up the stack and be handled by SERVER-LOOP. That means that an error triggered by client B will cause client A's handler to be unwound from the stack, aborting his connection in the process.

```
(defun error-propagation ()
  (with-clients (a b)
    (send a first-half)
    (send b "#@ <- illegal read syntax: will trigger an error.")
    (ignore-errors (send a second-half))))
```

The expected output on the server is:

```
;; Connection #2 aborted!
;; Connection #1 aborted!
;; Continuing after error: READER-ERROR
```

5 Conclusion

`SERVE-EVENT` presents a simple interface and makes it easy to write common server programs. However, you have to be thoughtful about how you use it, and be aware of what's on the stack. Otherwise you could be eaten alive on the big bad internet.