Custom Trail Camera Firmware Environment

Overview and Working Notes

2021-08-17 zak: Created

2021-08-31 zak: Pulled over to GoogleDrive; rethinking security

2021-09-15 zak: Need to update to reflect static (vs. dynamic) loading

# Introduction

# Firmware Architecture

## Principles

* Keep User Code separate, as much as possible, from base camera System Code
* Enable offline validation of user code prior to installation in camera
* Reduce the possibility that loading user code corrupts the baseline firmware and/or prevents the camera from rebooting, or updating firmware
* Security will not be any better than the security of the base camera

In keeping with first principle, the firmware is split into two images. The first image is an opaque (to the user) firmware load which consists primarily of the factory-installed firmware. Internal (“behind the firewall”) tools are used to add functions which will load the user code and data segments, provide an interface to library and service functions, and provide indirect function calls into the user binary.

The user binary is a standalone collection of program text and data created by an open source toolset. This binary is stored in a canonical file location in the internal flash file system for cold storage and is loaded into a well-defined location in DRAM on power up by the Modified Standard Firmware.

## System Firmware

The system firmware contains a set of opaque (to the user) additions to the factory firmware load needed to enable the user environment. These consist of:

* A set of service functions: Instead of exposing all of the firmware-defined functions to the user code, we create a new, well defined functional interface in the extended system firmware. Only functions in this interface are made available to the user code. These as library and service functions. This functional interface may call internal functions to implement the interface. All
* [TODO: what about bounds checking to make sure user doesn’t try to execute an index beyond the array?]
* An indirection table for user defined call-out functions: User code is invoked during the execution of the standard firmware image through an enumerated index into a table of pointers to user code. This table is located at the front of the user binary, and is loaded into memory with the rest of the binary image. Functions are accessed by an enum-based index into this table. Unused/unsupported user functions must be set to 0.
* A region in the system memory for storing data and text: this space is allocated during at compile time, and is the remainder of space made available by replacing a large, unused function in the original firmware with a stub, thus making the remainder of function’s footprint in memory available for system and user code and data segments. The system image is part of the EEPROM image, whereas the user code/data segments are loaded on system initialization from the file system. Before loading the user-defined binary file, the MSF performs a series of consistency checks. These shall include at least a check on the file size, and some sort of crc or checksum.
* A call to a user-defined initialization function at a well-defined location in the user text space. This function is called with arguments which give the location, in physical memory, of the user text and data segments.

## UI Structure

New functions set up as an extension to the “normal” menu? Or as a special menu mode?

* Special menu mode has the advantage of being completely invisible to the normal operation of the camera
* Special menu mode is a little cumbersome to enter.

Conclusion: could do both. Let’s start by seeing if we can just extend the existing menu

Menu Structure

Extend the initial 3-item menu (Setup, Playback, Home) with one more item, “Custom”. Build one layer of menu below this. Candidates:

* Operating Hours
* DSLR Trigger
* White Flash
* Battery Monitor
  + Extend Battery type to NiMH

Menu mode vs. auto mode: we want to support some functions which execute when the camera is in the field (e.g. on\_photo and on\_video). But it seems like a good design principle to keep the custom UI separated from these.

## User Binary Image

# Offline Validation

Because of the low level nature of the camera firmware environment, we develop a set of tools and restrictions to provide some validation of user code before it is installed on the cameras.

## Emulation

Dynamic Program Checking

No pointers outside of the malloc’ed region

All user functions must exit and return within a bounded number of instructions. This is only a solvable problem if we restrict user programs to accept no arguments or variable parameters. This is overly restrictive. Instead, we will make sure that user programs terminate for a randomized set of inputs and parameters.

## Argument and Parameter Randomization