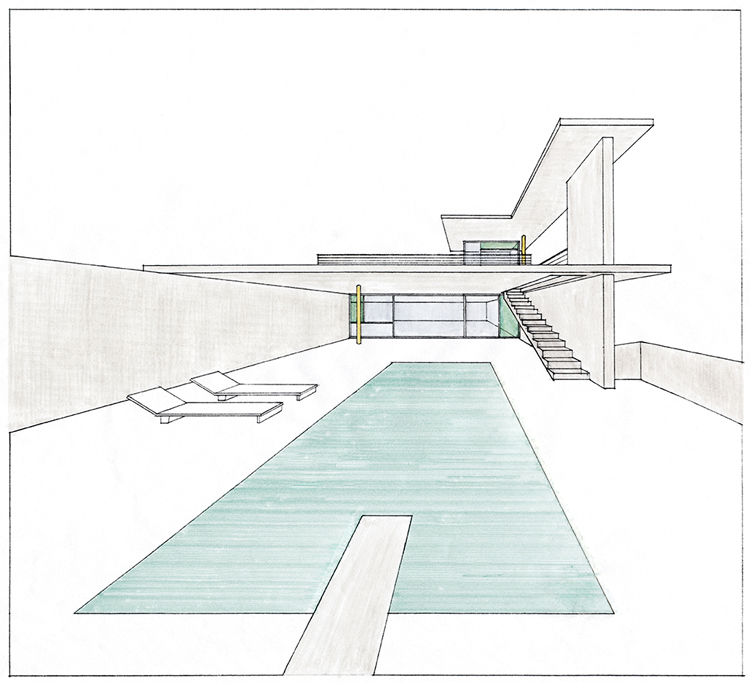
Distributed Binary Fuzzing

Prepared for: Publication

Prepared by: Michael Guidry, CEO [[mike@unifieddefense.net](mailto:mike@unifieddefense.net)]

Unified defense technologies, inc.

DATE \@ "MMMM d, y" October 23, 2015

Proposal number: 1

# EXECUTIVE SUMMARY

## Objective

Increase security for the popular applications, and protocols used on the internet. Fuzzing applications usually work around file interpreters. It requires re-execution of the same software over, and over hoping that a crash occurs. The debugger then has to determine if the crash can gain control of the applications execution flow. The software that this document discusses intends to automate the process, and allow deep binary fuzzing of every portion of the application. This includes TCP/IP communications as well. It is built in different aspects to support the mechanism of scaling across HPC machines.

## Goals

Lower the cost of exploitation by finding bugs on a mass scale using a distributed system. Find bugs otherwise never to be reached using traditional methods. Allow more in tune knowledge of internal process execution flow before, during, and after a bug is triggered. The security researcher could use this information to develop the exploit at an alarming rate without too many sleepless nights.

## Solution

Break the tasks into different sections. Use emulation. Distribute it across a farm of HPC servers. Operating system agnostic although the PoC is for Win32. It could easily fuzz the kernel using the same technique as well.

Unified defense technologies, inc.

## Project Outline

The program has several core systems which are all highly important. Each one individually isn’t so interesting, however together they allow fuzzing at a rate unknown publicly.

* Analysis of the application’s core code, as well as any libraries used. This includes disassembling instructions, creating a list of functions, outlining OS level libraries where fuzzing data will be injected, and emulating to detect software development errors.
* Execute the application normally until a function (such as ReadFile, Winsock recv(), etc) happens, and then keep information regarding where the data will be stored. Dump all memory from this process including thread context, module information, and all memory regions. This creates a ‘snapshot’ of a particular portion of the application to fuzz, and then you can remove the breakpoint and repeat.
* Load this snapshot into the emulator on a Linux cluster. It should execute the software at least one attempt on the master server, and then calculate how to spit up the task for a cluster. It needs to have its own virtual memory, CPU emulator, etc. A fully functional emulator is necessary.
* Start an API Proxy server. This server will handle the OS level API that we do not want to emulate due to resources, and being irrelevant. The API proxy communication with the emulator will get logged into a communication exchange list. This allows the cluster to retrieve all of the same values after it has only happened once. This is a huge portion of the magic of allowing the system to execute, and fuzz the application properly.

company name

# BUDGET

## Time

I have already spent random blocks of days, and nights over the course of the past several years. The code base is around 10,000 lines for the analysis, and emulator engine alone. I am looking to find some people in the security community that would like to help with the project via github, or another version control system.

## Reward

Recognition. I’m unsure where the project will go however it is the only truly scalable fuzzing system which does not waste CPU cycles on useless aspects of the software in question. Fuzzing of this nature will purely execute the instructions that are important to the particular data, and then complete its task. I’m not going to calculate out specifically how much of a speed increase this is. I plan on executing this on a 10,000 core cluster shortly. A data interpreter inside of an application is usually only a few functions. It however takes the majority of its time for process creation, destroying, library loading, file system IO, etc. The virtual memory strategy has several features that are expressed more in the code, and other comments.