FIRMWARE SLAP: AUTOMATING DISCOVERY OF EXPLOITABLE VULNERABILITIES IN FIRMWARE

CHRISTOPHER ROBERTS
WHO AM I

• Researcher at REDLattice Inc.
• Interested in finding bugs in embedded systems
• Interested in program analysis
• CTF Player
A QUICK BACKGROUND IN EXPLOITABLE BUGS
DARPA CYBER GRAND CHALLENGE

• Automated cyber reasoning systems:
  • Find vulnerabilities
  • Exploit vulnerabilities
  • Patch vulnerabilities

• Automatically generates full exploits and proof of concepts.
PREVENTING BUGS AUTOMATICALLY

- Source level protections
  - LLVM’s Clang static analyzers
- Compile time protections
  - Non-executable stack
  - Stack canaries
  - RELRO
  - _FORTIFY_SOURCE
- Operating system protections
  - ASLR
PREVENTING BUGS AUTOMATICALLY In Embedded Devices

- Source level protections
  - LLVM’s Clang static analyzers - Maybe
- Compile time protections
  - Non-executable stack - Maybe
    - Stack canaries
    - RELRO
    - _FORTIFY_SOURCE
- Operating system protections
  - ASLR
There has to be an exploit to mitigate it, right?
<table>
<thead>
<tr>
<th>Non-executable stack</th>
<th>Stack Canaries</th>
<th>RELRO</th>
<th>_FORTIFY SOURCE</th>
<th>ASLR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Router 1" /></td>
<td>✅</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image2" alt="Router 2" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image3" alt="Router 3" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image4" alt="Router 4" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image5" alt="Router 5" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image6" alt="Router 6" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image7" alt="Router 7" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image8" alt="Router 8" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image9" alt="Router 9" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image10" alt="Router 10" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><img src="image11" alt="Router 11" /></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
DEMO

- CVE-2019-13087
- CVE-2019-13088
- CVE-2019-13089
- CVE-2019-13090
- CVE-2019-13091
- CVE-2019-13092

Mom come pick me up I’m scared
CONCOLIC ANALYSIS

- Symbolic Analysis + Concrete Analysis
  - Lots of talks already on this subject.
  - Really good at finding specific inputs to trigger code paths.
  - For my work in Firmware Slap I used angr!
    - Concolic analysis
    - CFG analysis
    - Used in Cyber Grand Challenge for 3rd place!
BUILDING REAL INPUTS FROM SYMBOLIC DATA

- Symbolic Variable Here
  - get_user_input()
  - To get our “You did it” output
    - angr will create several program states
      - One has the constraints:
        - x >= 200
        - x < 250
      - angr sends these constraints to it’s theorem prover to give:
        - X=231 or x=217 or x=249...

```c
int x = get_user_input();
if (x >= 200)
{
    if (x < 250)
    {
        printf("You did it!\n");
    }
    else
    {
        printf("WRONG\n");
    }
}
else
{
    printf("WRONG\n");
}
```
• Symbolically represent more of the program state.
  • Registers, Call Stack, Files

• Query the analysis for more interesting conditions
  • Does a network read influence or corrupt the program counter?
  • Does network data get fed into sensitive system calls?

• Can we track all reads and writes required to trigger a vulnerability?
WHERE DOES CONCOLIC ANALYSIS FAIL?

- **Big code bases**
- **Angr** is trying to map out every single potential path through a program. Programs of non-trivial size will eat all your resources.
- A compiled lighttpd binary might be ~200KB
- Angr will run your computer **out of memory** before it can example every potential program state in a webserver
- Embedded system’s firmware can be a lot larger…
• Challenge:
  • Model complicated binaries with limited resources
  • Model unknown input
  • Identify vulnerabilities in binaries
  • Find binaries and functions that similar to one-another
• Underconstraining concolic analysis:
  • Values from hardware peripherals and NVRAM are **UNKNOWN**
  • Spin up and initialization consumes valuable **time** and **resources**
  • Configs can be setup any number of ways

• **Skip the hard stuff**
  • Make hardware peripherals and NVRAM return **symbolic** variables
  • Start concolic analysis **after** the initialization steps
Start

Parse Config

Setup sockets

Parse user input

angr can analyze code at this level, but it needs to know where to start.

Ghidra can produce a function prototype that angr can use to analyze a function...
• Finding bugs in binaries
  • Recover every function prototype using ghidra
  • Build an angr program state with information with symbolic arguments from the prototype
  • Run each analysis job in parallel
FINDING BUGS IN FUNCTIONS

- Demo
• With less code to analyze we can introduce more heavy-weight analysis
  • Tracking memory instructions imposed by all instructions
  • Memory regions tainted by user supplied arguments
  • Mapping memory loading actions to values in memory.

• Every step through a program
  • Store any new constraints to user input
  • Does user input influence a system() call or corrupt the program counter
  • Does user input taint a stack or heap variable
void remove_routing_rule(char *iptable_arg, char *netmask_arg, char *inet_device_arg)
{
  int iVar1;
  size_t buf_len;
  size_t sVar2;
  size_t buf_len2;
  char *user_buf;
  char acStack276 [256];

  strncpy(&user_buf,"route del ",0x100);
  iVar1 = strcmp(netmask_arg,"255.255.255.255");
  if (iVar1 == 0) {
    buf_len = strlen((char *)&user_buf);
    /* -host */
    *(undefined4 *)&user_buf + buf_len = 0x736f682d;
    acStack276[buf_len] = 't';
    acStack276[buf_len + 1] = '\';
    acStack276[buf_len + 2] = 0;
  } else {
    buf_len2 = strlen();
    /* -net */
    *(undefined4 *)&user_buf + buf_len2 = 0x7465626d;
    acStack276[buf_len2] = 't';
    acStack276[buf_len2 + 1] = '\';
    acStack276[buf_len2 + 2] = 0;
  }
  strcat((char *)&user_buf,iptable_arg);
  sVar2 = strlen((char *)&user_buf);
  *(undefined *)&user_buf + sVar2 = 0x20;
  *(undefined *)&user_buf + sVar2 + 1 = 0;
  iVar1 = strcmp(netmask_arg,"255.255.255.255");
  if (iVar1 != 0) {
    snprintf((char *)&user_buf,0x100, "%s netmask %s",&user_buf,netmask_arg);
  }
  snprintf((char *)&user_buf,0x100, "%s dev %s ",&user_buf/inet_device_arg);
  system((char *)&user_buf);
  return;
}
void remove_routing_rule(char *iptable_arg, char *netmask_arg, char *inet_device_arg)
{
    int iVar1;
    size_t buf_len;
    size_t buf_len2;
    char *user_buf;
    char acStack276[256];

    strlcpy(&user_buf,"route del  ",0x100);
    iVar1 = strcmp(netmask_arg,"255.255.255.255");
    if (iVar1 == 0) {
        buf_len = strlen((char *)&user_buf);
        /* -host */
        *(undefined *)((int *)&user_buf + buf_len) = 0x736f682d;
        acStack276[buf_len] = 't';
        acStack276[buf_len + 1] = ' ';
        acStack276[buf_len + 2] = 0;
    } else {
        buf_len2 = strlen();
        /* -net */
        *(undefined *)((int *)&user_buf + buf_len2) = 0x7465732d;
        acStack276[buf_len2] = 't';
        acStack276[buf_len2 + 1] = 0;
    }
    strcat((char *)&user_buf,iptable_arg);
    iVar1 = strlen((char *)&user_buf);
    *(undefined *)(((int *)&user_buf + iVar1) - 0x20);
    *(undefined *)(((int *)&user_buf + iVar1 + 1) - 0);
    iVar1 = strcmp(netmask_arg,"255.255.255.255");
    if (iVar1 != 0) {
        snprintf((char *)&user_buf,0x100,"%s netmask %s", &user_buf,netmask_arg);
    }
    snprintf((char *)&user_buf,0x100,"%s dev %s ", &user_buf,inet_device_arg);
    system((char *)&user_buf);
    return((char *)&user_buf);
}
void remove_routing_rule(char *iptable_arg, char *netmask_arg, char *inet_device_arg)
{
    int iVar1;
    size_t buf_len;
    size_t sVar2;
    size_t buf_len2;
    char *user_buf;
    char acStack276 [256];

    strncpy(&user_buf,"route del ",0,00);
    iVar1 = strlen((char *)&user_buf);
    /* -host */
    *(undefined4 *)((int)&user_buf + buf_len) = 0x736f682d;
    acStack276[buf_len] = 't';
    acStack276[buf_len + 1] = ' ';
    acStack276[buf_len + 2] = 0;
}
else {
    buf_len2 = strlen();
    /* -net */
    *(undefined4 *)((int)&user_buf + buf_len2) = 0x74656e2d;
    acStack276[buf_len2] = 't';
    acStack276[buf_len2 + 1] = 0;
}

    strcat((char *)&user_buf, iptable_arg);
    sVar2 = strlen((char *)&user_buf);
    *(undefined *)((int)&user_buf + sVar2) = 0x20;
    *(undefined *)((int)&user_buf + sVar2 + 1) = 0;
    iVar1 = strcmp(netmask_arg,"255.255.255.255");
    if (iVar1 != 0) {
        snprintf((char *)&user_buf,0x100, "%s netmask %s", &user_buf, netmask_arg);
    }
    snprintf((char *)&user_buf,0x100, "%s dev %s ", &user_buf, inet_device_arg);
    system((char *)&user_buf);
    return;
}
Command Injection found in internet.cgi at remove_routing_rule

void remove_routing_rule(char *iptable_arg, char *netmask_arg, char *inet_device_arg)
{
    int iVar1;
    size_t buf_len;
    size_t sVar2;
    size_t buf_len2;
    char *user_buff;
    char acStack276 [256];

    strncpy(&user_buff,"route del -host 'reboot' @@@ @@ dev @@ @@
    iVar1 = strncmp(netmask_arg, "255.255.255.255\n    if (iVar1 == 0) {
        buf_len = strlen((char *)&user_buff);
        "-host */
        *(undefined *)((int *)&user_buff + buf_len) = 0x736f682d;
        acStack276[buf_len] = 't';
        acStack276[buf_len + 1] = '\n        acStack276[buf_len + 2] = 0;
    }
    else {
        buf_len2 = strlen();
        "-net */
        *(undefined *)((int *)&user_buff + buf_len2) = 0x746562ed;
        acStack276[buf_len2] = '\n        acStack276[buf_len2 + 1] = 0;
    }
    strcat((char *)&user_buff, iptable_arg);
    sVar2 = strlen((char *)&user_buff);
    *(undefined *)((int *)&user_buff + sVar2) = 0x20;
    *(undefined *)((int *)&user_buff + sVar2 + 1) = 0;
    iVar1 = strncmp(netmask_arg, "255.255.255.255\n    if (iVar1 != 0) {
        snprintf((char *)&user_buff, 0x100, "%s netmask %s", &user_buff, netmask_arg);
    }
    snprintf((char *)&user_buff, 0x100, "%s dev %s ", &user_buff, inet_device_arg);
    system((char *)&user_buff);
    return;
}
Injected Memory Location

0x7ffeefed8 -> b'route del -host `reboot`
FUNCTION SIMILARITY

- Bindiff and diaphora are the standard for binary diffing.
- They help us find what code was actually patched when a CVE and a patch is published.
- Uses a set of heuristics to build a signature for every function in a binary
  - Basic block count
  - Basic block edges
  - Function references
• Both of these tools are tied to IDA

• The workflow is built around one-off comparisons
CLUSTERING

• Helps us understand how similar are two things?

• Extract features from each thing
  • For dots on a grid it can be:
    • X location
    • Y location
K-MEANS CLUSTERING

1. Extract features
2. Pick two random points
3. Categorize each point to one of those random points
   • Use Euclidian or cosine distance to find which is closest
4. Pick new cluster center by averaging each category by feature and using the point closest.
5. Recategorize all the points into categories.
   • Rinse and repeat until points don’t move!
CLUSTERING – WHY THIS WORKS

- **Features** don't have to be numbers...
- They can be the existence (0 or 1) of:
  - String references
  - Data references
  - Function arguments
  - Basic block count
- All of these features can be extracted from reverse engineering tools like...
  - Ghidra, Radare2, or Binary Ninja
IT ONLY WORKS IF YOU GUESS THE RIGHT NUMBER OF CLUSTERS
SUPERVISED CLUSTERING

- Supervised anything machine learning uses KNOWN values to cluster data.
- We also know how many clusters there should be.
- Our functions inside our binaries could be supervised if every function was known to be vulnerable or benign.
- Embedded systems programming gives us no assurances.
SEMI-SUPERVISED CLUSTERING

• Semi-Supervised clustering uses SOME KNOWN values to cluster data.
• If we use public CVE information to find which functions in a binary are KNOWN vulnerable, we can guess that really similar functions might also be vulnerable.
• We can set our cluster count to the number of known vulnerable functions in a binary.
• **Finding features** in binaries to cluster
  • Wrote a Ghidra headless plugin to dump all function information
  • Data/String/Call references are changed to binary (0/1) it exists or it doesn’t
  • All numbers are **normalized**
    • Being at offset 0x80000000 shouldn’t matter more then having 2 function arguments.
  • **Throw away useless information**
    • A Chi^2 squared test is used to see how much a feature defines an item.
    • If every function has the same calling convention, the Chi^2 squared test will **throw it away**.
• Taking it further…
  • Selecting a better number of clusters through cluster scoring
  • Silhouette score ranks how similar each cluster of functions are

• This separates functions into clusters of similar tasks
  • String operation functions
  • Destructors/Constructors
  • File manipulation
  • Web request handling
  • etc..
SILHOUETTE SCORE
SILHOUETTE SCORE
DATA MINING + CONCOLIC ANALYSIS

• Demo
• CVE-2019-13087
<table>
<thead>
<tr>
<th>Similar Functions With Distances</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>sym.mtd_write_firmware in upload_bootloader.cgi</td>
<td>0.0</td>
</tr>
<tr>
<td>sym.mtd_write_firmware in upload.cgi</td>
<td>1.868202467580904e-05</td>
</tr>
<tr>
<td>sym.mtd_write_bootloader in upload_bootloader.cgi</td>
<td>0.000152721111900810348</td>
</tr>
<tr>
<td>sym.mtd_write_bootloader in upload.cgi</td>
<td>0.00016208252348248742</td>
</tr>
<tr>
<td>sym.write_flash_kernel_version in upload.cgi</td>
<td>0.013812828298550572</td>
</tr>
<tr>
<td>sym.write_flash_kernel_version in upload_bootloader.cgi</td>
<td>0.001381969125964222</td>
</tr>
<tr>
<td>sym.access_policy_handle in wireless.cgi</td>
<td>0.0030522430530255384</td>
</tr>
<tr>
<td>sym.compile_regex in wireless.cgi</td>
<td>0.003158924286679299</td>
</tr>
</tbody>
</table>

- **Findings**
  - Code clones
  - Calling patterns
  - Similar function calls
  - Similar data references
  - Similar file access
```c
int mtd_write_firmware(int param_1, int param_2, char *param_3)
{
    int uVar1;
    char sys_buffer[516];
    snprintf(sys_buffer, 0x200, 
             "/bin/mtd_write -o %%d -l %%d write %%s Kernel", param_2, 
    uVar1 = system(sys_buffer);
    return uVar1;
}

void mtd_write_bootloader(int param_1, int param_2, char *param_3)
{
    char sys_buf[516];
    snprintf(sys_buf, 0x200, 
             "/bin/mtd_write -o %%d -l %%d write %%s Bootloader", param_2, 
    printf("write bootloader");
    system(sys_buf);
    return;
}
```

Similar Functions With Distances

<table>
<thead>
<tr>
<th>Function</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtd_write_firmware</td>
<td>in &gt; 0.0</td>
</tr>
<tr>
<td>mtd_write_firmware</td>
<td>in &gt; 1.868202467580904e-05</td>
</tr>
<tr>
<td>mtd_write_bootloader</td>
<td>in &gt; 0.00015272111900810348</td>
</tr>
</tbody>
</table>
The trick is generally finding the biggest "drop" and choosing the count before that.
FINDING THE FUNCTION CLUSTER COUNT

• The trick is generally finding the biggest "drop" and choosing the count before that.
FIRMWARE SLAP

- Extract Firmware
  - Locate System root
  - Recover Function prototypes from every binary
    - Build and run angr analysis jobs
      - Extract Best function features using SKlearn
        - Cluster Functions according to best feature set
          - Export Data to JSON and send into elastic search
VISUALIZING VULNERABILITY RESULTS

- All information generated as JSON from both concolic and data mining pass
- Includes script to load information into Elasticsearch and Kibana
MITIGATIONS

- Use compile time protections
- Enable your operating system’s ASLR
- Buy a better router
• It’s time to bring more automation into checking our embedded systems
• Don’t blindly trust third-party embedded systems
  • I’m giving you the tools to find the bugs yourself
RELEASING

- **Firmware Slap** – The tool behind the demos
- The **Ghidra** function dumping plugin
- The cleaned-up **PoCs**
• Code:
  • https://github.com/ChrisTheCoolHut/Firmware_Slap

• Feedback? Questions?
  • @0x01_chris