This paper appeared in *1st Workshop on the Security of Cyber Physical Systems* (WOS-CPS), Vienna, Austria, September 2015.

Umbra: Embedded Web Security through Application-Layer Firewalls

Travis Finkenauer and J. Alex Halderman

University of Michigan
tmfink@umich.edu
jhalderm@umich.edu

Abstract Embedded devices with web interfaces are prevalent, but, due to memory and processing constraints, implementations typically make use of Common Gateway Interface (CGI) binaries written in low-level, memory-unsafe languages. This creates the possibility of memory corruption attacks as well as traditional web attacks. We present Umbra, an application-layer firewall specifically designed for protecting web interfaces in embedded devices. By acting as a "friendly man-in-the-middle," Umbra can protect against attacks such as cross-site request forgery (CSRF), information leaks, and authentication bypass vulnerabilities. We evaluate Umbra's security by analyzing recent vulnerabilities listed in the CVE database from several embedded vendors and find that it would have prevented half of the vulnerabilities. We also show that Umbra comfortably runs within the constraints of an embedded system while incurring minimal performance overhead.

Keywords: embedded security, firewall, web security

1 Introduction

Embedded devices such as routers [16], printers [26], and supervisory control and data acquisition (SCADA) systems [8, 53] are frequently managed through web interfaces, which potentially create an opening for remote attackers. Many of these systems are critical, such as SCADA systems that manage utilities and medical devices that support life directly [22]. These web interfaces often use Common Gateway Interface (CGI) binaries implemented in low-level languages, such as C, which introduces the possibility of memory corruption attacks and input validation vulnerabilities. In one recent example, researchers found that an embedded web interface's login page had remotely exploitable buffer overflow vulnerabilities in the username and password fields, which would allow an attacker to take control over the host system [6]. The same implementation was also vulnerable to several other textbook attacks, including shell injection and authentication bypass.

However, this is not a case of a single careless vendor that makes products with vulnerabilities; embedded devices in general tend not to have strong security.

With the recent Misfortune Cookie vulnerability, over 200 models of routers from several vendors were found to be using an out-of-date web server that was vulnerable a memory corruption attack, allowing an attacker to gain complete control over the router [9]. Many vendors were using a decade-old version of the RomPager web server in their device firmware. Problems like these are all too common, and, coupled with the fact that embedded devices can act as a foothold into a target network [29], they make embedded web interfaces an attractive attack vector for prospective intruders.

Ideally, all the code in an embedded device would be subjected to a security audit. However, such audits are expensive and time consuming, require security expertise, and may not be feasible when embedded devices reuse off-the-shelf closed-source components [43]. As an alternative, we propose introducing a small layer of security software that can be integrated into many kinds of embedded devices and act as a "friendly man-in-the-middle" that enforces a security policy set by the developers of each device. Such an application-layer firewall can provide concentrated protection at the web interface's attack surface and greatly reduce overall vulnerability at low cost.

We present Umbra, our implementation of an application-layer firewall that can be easily integrated with preexisting embedded systems to provide additional security. Umbra is designed to work with existing web server binaries, to be simple to configure, and to work within the limited resources of embedded systems.

In order for a manufacturer-side security solution to garner adoption, it must have small perceived cost to developers. For example, enabling stack canaries only requires a compiler flag and has gained wide adoption; today's compilers often enable stack canaries by default [13, 57]. In contrast, various heavy-weight approaches, such as dynamic taint tracking have not gained traction, because they introduce on the order of two times slowdown [7, 15]. Since Umbra acts as a "shim" in front of existing web servers, manufacturers can keep their existing code and thus reduce the cost to adopting security. Also, embedded devices' code is often licensed from third-party vendors, so the embedded developer does not always have the source code for all parts of the firmware. Umbra works on systems where source code is not available.

Umbra needs to have information about the web application being protected. We define a policy language that allows embedded system developers to easily describe security properties for an HTTP interface, such as the set of allowed characters or maximum length for HTTP parameters. The Umbra shim is compiled together with these policies and enforces the specified properties.

We evaluate Umbra's functionality and performance. In a review of recent vulnerabilities in some of Umbra's target devices, we find that Umbra would have prevented or mitigated more than half of these issues. We also show that when running on a Raspberry Pi system with OpenWrt, Umbra adds only 5 ms (about 3%) to average page download times, demonstrating that it comfortably runs within the resource constraints of embedded Linux systems.

Source Code Release We are releasing our Umbra prototype as open-source software. It is available at https://github.com/umbra-firewall/umbra.

2 Related Work

There are various standalone application-layer firewalls that secure HTTP interfaces, including Barracuda Web Application Firewall [3], Cisco ACE Web Application Firewall [10], and HP TippingPoint [27]. Some of these devices even provide features such as outbound filtering of sensitive data, including credit card and social security numbers, which can aid organizations in being compliant with the Payment Card Industry Data Security Standard (PCI DSS) [47].

IronBee and ModSecurity are examples of host-based application layer firewalls. ModSecurity supports Apache, Nginx, and Microsoft IIS web servers, and IronBee supports Apache and Nginx [48,56]. These web servers are not commonly used in embedded systems because of their larger CPU, memory, and storage footprint compared to special-purpose embedded servers. Instead, embedded devices use servers such as lighttpd [33], uhttpd [45], or a custom HTTP server (e.g., [37]). Hence, these existing solutions are not appropriate for integration with embedded devices.

Many recent studies have demonstrated vulnerabilities in embedded devices—including traffic light cameras [23], server lights-out management controllers [6], and automobile controller-area network (CAN) buses [50], to name just a few—and the trend towards the "Internet of Things" suggests that there will be vastly more embedded devices in the future [36]. Many embedded devices get connected to the public Internet, where they can be mapped and probed with publicly available tools [19]. In one case, a vulnerability in just two UPnP libraries affected 2% of public IPv4 addresses [42]. The security of embedded devices is a significant and growing issue, and the widespread vulnerability seen among today's devices demonstrates that even basic network protections are often absent.

3 Design and Implementation

Embedded systems may often be administered through a variety of protocols, including HTTP, SSH, Telnet, IPMI, and proprietary protocols [25, 26, 31, 55]. HTTP also sometimes acts as the transport for other protocols (such as SOAP [34] and HNAP [11]) and REST style interfaces [46]. Since HTTP is widely used and subject to a range of common vulnerabilities, Umbra focuses on defending web interfaces, though the architecture could be extended to support other protocols.

In this work, we only consider attacks targeting embedded web interfaces, through either HTTP or HTTPS. Our threat model assumes that the attacker does not have valid credentials or physical access to the target. For example, the attacker can submit arbitrary content to forms or visit arbitrary URLs corresponding to the targeted device. We also consider attacks where a legitimate user may visit an attacker-controlled web page to allow for cross-site request forgery (CSRF) attacks.

Umbra works by acting as a transparent proxy between clients and the web server, as illustrated in Figure 1. The pass-through is achieved by introducing a small software layer, which we call the Umbra *shim*, that runs on the embedded

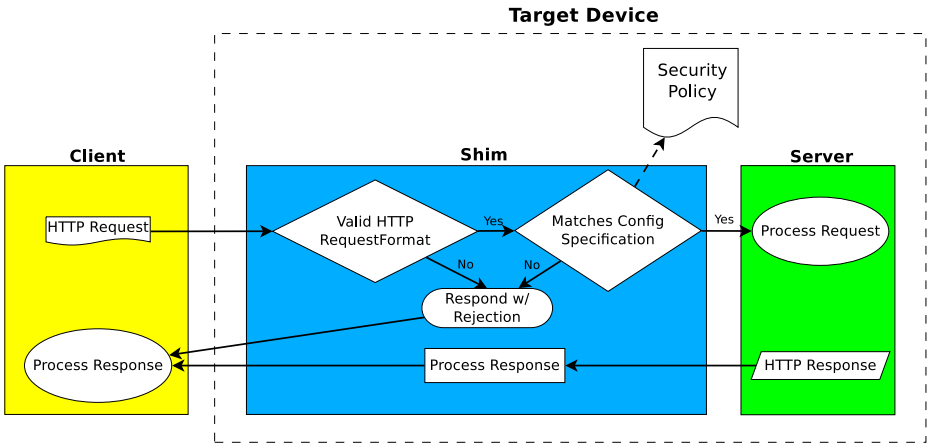

Figure 1. Umbra Architecture—A lightweight software "shim" runs on the embedded device and transparently proxies client requests, rejecting those that are malformed or that do not match a predefined security policy.

device and listens on the standard HTTP or HTTPS port. When the Umbra shim receives client requests, it checks to ensure that they comply with a *security policy*. This policy, defined in advance by the device manufacturer, specifies which security features should be applied and how the features should be enforced. For example, the policy might dictate the maximum allowed length for HTTP form inputs and specify a whitelist of permitted characters. (For details, see Section 3.2.) Umbra forwards requests that satisfy the policy to the original embedded web server and responds with rejections to requests that do not.

Umbra is designed to be integrated into device firmware by the manufacturer. To apply Umbra to an existing device, the manufacturer must:

1. Write a security policy tailored to the device's embedded web application.
2. Compile the Umbra shim for the embedded device. The policy gets compiled into the Umbra binary.
3. Configure the device's existing web server to listen on an alternate port and on the loopback interface.
4. Set the Umbra shim to run at boot time, listening on the default web port on the external interface.

Note that a device maker can add Umbra to its firmware without having to modify the source code for the existing web application.

3.1 Security Features

Our Umbra implementation is designed to protect against a range of common attacks, as summarized in Table 1. By adjusting the global settings in the security policy, it can be configured to have any combination of the security features described below:

Vulnerability	Security Features
XSS	Parameter whitelist
CSRF	CSRF protection
Authentication bypass	Authentication enforcement HTTP method whitelist
Information leak	Authentication enforcement HTTP method whitelist Directory traversal check
CGI memory corruption	Parameter character whitelist Parameter length check Header field length limit
Directory traversal	Directory traversal check

Table 1. Umbra Security Features — Umbra's security features mitigate or protect against a variety of common attacks.

CSRF Protection Cross-site request forgery (CSRF) attacks occur when an attacker causes a victim to perform an unintended action while the victim is logged into a target website [4]. For example, the victim might visit an attacker-controlled web page that uses JavaScript to make a POST request to http://target-site.com/delete-account. Since the victim's browser is logged in to the target site, the request includes the victim's session cookies, and the site accepts this request as authorized.

Umbra prevents CSRF attacks by using a CSRF-prevention token [4], a well-known technique used by many web frameworks [5,17]. To implement this defense, Umbra generates a random CSRF-prevention token for each browser session, which it sends as a cookie in every HTTP response. The shim also modifies pages that are specified in the security policy as requiring CSRF protection. It injects JavaScript into these pages that modifies HTML forms to add a hidden field containing the same token. For pages that the security policy specifies as receiving a form action from a CSRF-protected form, Umbra verifies that the submitted data contains a token that matches the one in the client's cookie.

Pages that both submit and receive a CSRF protected form (e.g., pages with self-referencing forms) present a complication, since a client would never be able to navigate directly to the page with the correct CSRF token. For such pages, the shim only enforces the presence of a CSRF token parameter for HTTP POST requests, not GET requests.

Although this approach requires client browsers to have JavaScript enabled, this is a reasonable assumption, as many embedded web interfaces require JavaScript (e.g., [37, 55]). A downside to this technique is that it only works when HTML forms send requests, and hence would not work when other methods are used to send HTTP requests, such as JavaScript's `XMLHttpRequest` or browser plug-ins. Device manufacturers would need to manually implement

CSRF protection for these methods. We note that, as with other common CSRF defenses, an attacker could bypass the protection by exploiting a separate XSS vulnerability [12].

Page-level Authentication Enforcement A common vulnerability in embedded devices is where page-level authentication is not properly enforced. For example, in a set of five simultaneously-disclosed vulnerabilities found in D-Link IP cameras, four were due to incorrect authentication [49]. With another brand of IP camera, an attacker could visit most pages without any authentication [18].

To protect against such vulnerabilities, the Umbra shim enforces RFC 2617 HTTP Basic Authentication [21]. The suggested configuration would be to require authentication by default and specify pages that do not require authentication, such as a status page or login page. The shim reads credentials from a file specified as a command-line parameter. This file is reloaded on each page request, so the web application may change the credentials by modifying this file at run time.

Directory Traversal Protection Another common class of attacks involves directory traversal vulnerabilities (e.g., [37]), where an attacker controls part of a file path and can inject a relative path to escape a directory. For example, an attacker may be able to inject a path such as "`../../passwords.txt`" to read the password file stored two directories above. To guard against this, the Umbra shim blocks HTTP requests with URLs that contain two periods in a row ("`..`").

The directory traversal protection does not currently attempt to check for directory traversal in HTTP parameters. Future versions of Umbra might add this as a per-parameter option, although this would introduce further complexity to the security policies.

HTTP Method Whitelist HTTP allows for pages to be accessed via different methods, such as GET, POST, or HEAD. Different web applications may not account for pages being viewed with an unexpected method and may leak information unintentionally. In order to prevent this, Umbra security policies can specify which HTTP methods are allowed for each page.

HTTP Header Length Limits The HTTP protocol includes headers that provide metadata about requests and responses, such as the client's user agent or the language of a response. Maliciously crafted headers have been used to exploit buffer overviews and command injection vulnerabilities in embedded web applications [14]. Umbra can be configured to limit the length of both HTTP header fields and values. If the length of any field name or value exceeds the corresponding maximum length, the shim will block the request.

Per-parameter Limits There are two options that can be enforced on a per-HTTP-parameter basis. These protections can help mitigate various input validation vulnerabilities—such as shell injection, SQL injection, and cross-site scripting—as well as memory corruption attacks. The first is *character whitelists*,

which cause the shim to limit input characters to those from a specified set, such as lowercase letters and numbers. This is specified with a regular expression character class, such as `[a-z0-9]`. The second protection is *length enforcement*, in which the shim ensures that parameters do not exceed a given number of characters, such as the name and password field for a login page [6].

3.2 Security Policy Language

The embedded developer must specify the desired security policy for Umbra, and we provide a policy language for this purpose. The language is designed to make it easy to specify conservative settings globally and relax them for specific pages as necessary for compatibility. There are three sections in the configuration file: global configuration, default page policy, and per-page policy. Each section consists of a JSON object with the options specified as member pairs.

The `global_config` section includes directives for enabling and disabling security features. For example, to enable CSRF prevention, the user would add `"enable_csrf_protection": true`. This global configuration section also has global options that are not page specific, such as the maximum allowed header field lengths.

The `default_page_config` section sets the default policy for all pages. Each page-level option must have a default value specified here.

The `page_config` section sets policies for specific URI paths that override some or all of the default policy options. If the policy for certain options is not specified, then the default page policy will be used for those options. In this sense, each page inherits the default policy. For example, the default policy may be to enforce authentication for a page; however, the developer may want a status page to be visible without requiring authentication, so the developer could disable the authentication check for the status page.

In the per-page policy section, options for specific parameters can be specified, such as a character whitelist and length limits. This allows for parameter-level control of the security policy. For example, in Figure 2, the `favorite_vowel` parameter has a max length of six and only accepts characters that are vowels.

The configuration is interpreted with Python, which outputs C code that is compiled into the Umbra shim. With this technique, there does not need to be C code that interprets the configuration at run time, keeping the resulting binary smaller and the C code simpler.

We give a simple example of a security policy in Figure 2. The global configuration section enables several security features. The default page policy indicates that by default, GET and HEAD requests are allowed, authentication is required, no CSRF protection is used, and parameters are limited to a length of 30 and alphanumeric characters. There are two pages for which the default policy is overridden. For the root page (`/`), no authentication is required to view the page, and forms on the page will include the hidden CSRF token (see Section 3.1). The `/cgi-bin/favorites` page has a parameter that allows only vowels and another parameter that only allows numbers.

```
{
  "global_config": {
    "enable_request_type_check": true,
    "enable_param_len_check": true,
    "enable_param_whitelist_check": true,
    "enable_csrf_protection": true,
    "enable_authentication_check": true,
    "session_life_seconds": 300
  },
  "default_page_config": {
    "request_types": ["GET", "HEAD"],
    "requires_login": true,
    "has_csrf_form": false,
    "receives_csrf_form_action": false,
    "max_param_len": 30,
    "whitelist": "[a-zA-z0-9]"
  },
  "page_config": {
    "/":{
      "requires_login": false,
      "has_csrf_form": true
    },
    "/cgi-bin/favorites": {
      "request_types": ["POST"],
      "receives_csrf_form_action": true,
      "params": {
        "favorite_vowels": {
          "max_param_len": 6,
          "whitelist": "[aeiouy]"
        },
        "favorite_number": {
          "max_param_len": 3,
          "whitelist": "[0-9]"
        }
      }
    }
  }
}
```

Figure 2. Sample Umbra Security Policy — This is an example of a security policy file. Path-specific rules for `/` and `/cgi-bin/favorites` override the global defaults.

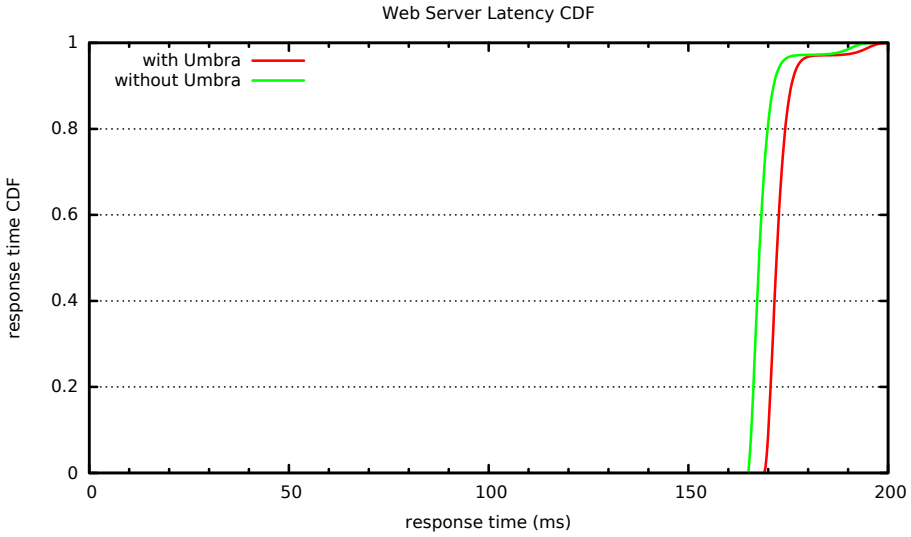

Figure 3. Server Response Time — This graph shows the response time of the web server with and without Umbra; the average overhead is about 5 ms (3%).

3.3 Implementation

We implemented our Umbra prototype for Linux, which is a popular platform for Internet-attached embedded devices. To provide high-performance non-blocking sockets, we use Linux's `epoll`(7) interface [20]. Since the shim needs to run within the footprint of embedded systems, we wrote it in C. We considered memory-safe languages such as Python and Go, but Python is not usually installed on embedded devices and requires a large runtime. While the Go compiler generates machine binaries, the Go runtime library is statically linked, leading to binaries that are prohibitively large.

The shim minimizes use of external libraries, only using an HTTP parsing library [30]. It also optionally links against OpenSSL to provide HTTPS support [44]. The complete Umbra implementation, including the shim and the policy interpreter, is 5676 lines of C and 631 lines of Python. When compiled as an ARM binary, the shim executable is 75 KB.

4 Evaluation

4.1 Performance

To measure the performance of Umbra, we used the Apache Benchmark tool, which records the time taken for an HTTP server to respond to requests [1]. We ran our benchmark on a Raspberry Pi Model B running OpenWrt Barrier Breaker 14.07. The requests were made from a laptop that was connected directly

Vulnerability	Protection Level				Total
	None	Partial	Full	Unknown	
Bypass	1	0	4	0	5
Command injection	0	0	2	0	2
CSRF	0	0	5	1	6
Directory traversal	0	0	1	0	1
Denial of service	3	1	1	1	6
Information leak	1	0	3	0	4
Memory corruption	2	0	1	0	3
Other	0	0	0	1	1
SQL injection	0	0	0	1	1
XSS	2	3	3	3	11
Protect Totals	9	4	20	7	40

Table 2. Evaluating Vulnerability Protection — We rate the level of protection Umbra would have provided against a sample of real-world vulnerabilities.

via a 100 Mbps Ethernet cable. We made 1000 requests to the web interface both with and without the Umbra shim.

The results of this benchmark are shown in Figure 3. Umbra, on average, added only about 5 ms of overhead, or about 3%. We configured Umbra to check header length, HTTP request method, directory traversal, and authentication. We had Apache Benchmark send the correct HTTP Basic Authentication credentials.

Umbra comfortably ran within the Raspberry Pi's footprint. The shim compiled to a 75 KB dynamically linked ARM binary. In comparison, the BusyBox binary in the firmware image is 370 KB. The OpenWrt firmware image itself is 40 MB, with 33 MB available, so the executable can easily fit within the firmware image. During the benchmark, the Raspberry Pi reported 2–3% CPU utilization and peaked at a virtual memory size of 1.2 MB.

4.2 Security

In order to estimate the security provided by Umbra, we surveyed vulnerabilities that were assigned Common Vulnerability and Exposure (CVE) identifiers since 2012 from eight embedded device manufacturers: Brother, Cannon, Cisco, D-Link, Linksys, Lorex, Netgear, Supermicro, TP-LINK, Trendnet, and Xerox. There were 284 vulnerabilities across all these vendors. We randomly selected 100 of them and classified each based on the information in the public CVE database and public exploit code [40].

For each vulnerability, we first determined whether the vulnerability was for an embedded web server. We found that 40 of the 100 vulnerabilities were related to embedded web interfaces. For each embedded web interface related vulnerability, we manually determined: the class of vulnerability, the Umbra feature that would mitigate or prevent the vulnerability (if any), and the level of protection provided by Umbra. We determined the level of protection based

Vulnerability	Security Feature						Total
	Auth	CSRF	Dir. Traver.	Header Length Check	Param. White.	Param. Length	
Bypass	4	0	0	0	0	0	4
Command injection	0	0	0	0	2	0	2
CSRF	0	5	0	0	0	0	5
Directory traversal	0	0	1	0	0	0	1
Denial of service	0	0	0	1	0	0	1
Information leak	2	1	0	0	0	0	3
Memory corruption	0	0	0	0	0	1	1
Other	0	0	0	0	0	0	0
SQL injection	0	0	0	0	0	0	0
XSS	0	0	0	0	3	0	3
Feature Totals	6	6	1	1	5	1	20

Table 3. Security Feature Applicability — We show how many times each Umbra security feature prevented different types of vulnerabilities in our sample.

on whether or not there exists an Umbra configuration that would prevent the vulnerability without breaking functionality. We classified the level of protection as either *none* (would provide no protection), *partial* (would prevent some parts of the vulnerability or make it more difficult to exploit), *full* (would completely eliminate the vulnerability), or *unknown* (there was not enough public information to determine whether Umbra could have prevented the vulnerability).

A summary of the types of vulnerabilities and the amount of protection provided by Umbra is shown in Table 2. At least half of the surveyed vulnerabilities would have been prevented by Umbra. Umbra would fully prevent 4 of the 5 authentication bypass vulnerabilities, 5 of the 6 CSRF vulnerabilities, and 3 of the 4 information leak vulnerabilities. This indicates that Umbra is an effective tool for mitigating these common classes of vulnerabilities.

Umbra would have worked less well for defending against denial of service and XSS vulnerabilities. Umbra only prevented 1 of the 6 denial of service vulnerabilities and 3 of the 11 XSS vulnerabilities in our sample. XSS vulnerabilities are difficult to prevent using a "pass-through" system like Umbra, because the system does not have context for the data—only the information provided in the security policy. Umbra's main defense against XSS vulnerabilities is the parameter character whitelist, where the attacker is limited by which characters may be passed through HTTP parameters. This may be enough to stop some XSS attacks but not all. For example, with an improperly escaped parameter that allows quotes, an attacker could close an HTML attribute and create a new attribute corresponding to a JavaScript callback. In other instances, XSS may be done through alternative vectors. For example, in CVE-2014-4645, an attacker could inject arbitrary HTML or JavaScript by altering the attacker machine's

hostname [39]. In some of these cases, the developer must properly escape the data in the web application to prevent the vulnerability.

The Umbra security features that would have prevented each of the sampled vulnerabilities are shown in Table 3. The authentication, CSRF prevention, and parameter whitelist features provided the largest security impact, accounting for 17 of the 20 vulnerabilities that would have been prevented by Umbra.

5 Future Work

In future work, we plan to investigate applying similar techniques to protect other management protocols commonly used in embedded devices, such as IPMI and SNMP. Both of these protocols are common in larger networks and provide a well-known attack surface [38, 41].

Future versions of Umbra might consider other implementation languages that offer better memory safety. One potential candidate is Rust, a systems-oriented language that provides memory safety and protection against other security bugs [51]. When we wrote the original version of Umbra, the Rust language was not yet stable, but there has since been a stable release [52].

An alternative to adopting a safer language would be to apply formal verification to the shim C code. Recent efforts have produced formally verified versions of an optimizing C compiler [35] and an operating system kernel with 8700 lines of C code [32], so verifying the 5617-line Umbra shim should be tractable.

One downside to Umbra is that the developer needs to define the security policy manually. Instead, the system could be extended to infer a security policy from examples of valid inputs during a "training" period, as is done by security modules such as AppArmor [2] and Grsecurity [54]. This would simplify integrating Umbra with existing devices and increase the likelihood of adoption.

Although Umbra mitigates several significant classes of vulnerabilities affecting embedded web interfaces, one common problem that it does not yet address is flawed TLS implementations. Embedded devices often do not use HTTPS, implement it incorrectly, or use self-signed or default TLS certificates [24]. Umbra could be extended to upgrade all connections to HTTPS and to automatically request a browser-trusted certificate from a robotic CA such as Let's Encrypt [28].

Lastly, to spur Umbra adoption and improve the security of embedded devices generally, we hope to work with embedded vendors to help them integrate Umbra into their products. If several manufacturers integrate Umbra, there will be competitive pressure for other vendors to adopt similar security mechanisms.

6 Conclusion

Web interfaces in embedded devices are a common source of security vulnerabilities. We have shown that Umbra, a light-weight application-layer firewall, can prevent about half of known vulnerabilities in embedded web interfaces while adding negligible run-time overhead. Unlike existing application-layer firewalls,

Umbra can run comfortably in the constrained memory and CPU footprint of an embedded device. Embedded systems present many security challenges, but Umbra offers a promising approach to helping them achieve defense-in-depth.

Acknowledgments

This material is based upon work supported by a gift from Super Micro Computer, Inc. We would particularly like to thank Arun Kalluri, Joe Tai, Linda Wu, Mars Yang, Tau Leng, and Charles Liang from Supermicro. Additional support was provided by the National Science Foundation under grants CNS-1345254, CNS-1409505, and CNS-1518888.

References

1. Apache Software Foundation: `ab` — Apache HTTP server benchmarking tool (Apr 2015), http://httpd.apache.org/docs/2.4/programs/ab.html
2. AppArmor Security Project: Getting Started (Sep 2011), http://wiki.apparmor.net/index.php/GettingStarted
3. Barracuda Networks: Barracuda web application firewall (2015), https://www.barracuda.com/products/webapplicationfirewall
4. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: 15th ACM Conference on Computer and Communications Security. pp. 75–88. CCS (2008)
5. Bigg, R., et al.: Ruby on Rails security guide (2015), http://guides.rubyonrails.org/security.html
6. Bonkoski, A., Bielawski, R., Halderman, J.A.: Illuminating the security issues surrounding lights-out server management. In: 7th USENIX Workshop on Offensive Technologies. WOOT (2013)
7. Bosman, E., Slowinska, A., Bos, H.: Minemu: The world's fastest taint tracker. In: 14th Int'l Conf. on Recent Advances in Intrusion Detection. RAID (2011)
8. Certec EDV: Atvise SCADA (2014), http://www.atvise.com/en/products-solutions/atvise-scada
9. Check Point: Misfortune cookie (Dec 2014), http://blog.checkpoint.com/2014/12/18/misfortune-cookie-the-hole-in-your-internet-gateway-3/
10. Cisco Systems: Cisco ACE web application firewall (May 2008), http://www.cisco.com/c/en/us/products/collateral/application-networking-services/ace-web-application-firewall/data_sheet_c78-458627.html
11. Cisco Systems: Home network administration protocol (HNAP) whitepaper (Jan 2009), http://www.cisco.com/web/partners/downloads/guest/hnap_protocol_whitepaper.pdf
12. Coen, T.: Bypass CSRF via XSS. Software talk (Mar 2015), http://software-talk.org/blog/2015/03/bypass-csrf-via-xss/
13. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: 7th USENIX Security Symposium (1998)
14. D-Link: DIR-645: Rev. Ax — Command injection — Buffer overflow: FW 1.04b12 (Jan 2015), http://securityadvisories.dlink.com/security/publication.aspx?name=SAP10051

15. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: A detection tool to defend against return-oriented programming attacks. In: 6th ACM Symposium on Information, Computer, and Communications Security. pp. 40–51. ASIACCS (2011)
16. DD-WRT Wiki: Web interface (Jul 2012), http://www.dd-wrt.com/wiki/index.php/Web_Interface
17. Django Software Foundation: Cross site request forgery protection (2015), https://docs.djangoproject.com/en/1.8/ref/csrf/
18. Doyle, J.: Lorex IP camera authentication bypass (CVE-2012-6451) (Dec 2012), https://www.fishnetsecurity.com/6labs/blog/lorex-ip-camera-authentication-bypass-cve-2012-6451
19. Durumeric, Z., Wustrow, E., Halderman, J.A.: ZMap: Fast Internet-wide scanning and its security applications. In: 22nd USENIX Security Symposium (2013)
20. `epoll`(7): process trace. Linux Programmer's Manual
21. Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: HTTP authentication: Basic and digest access authentication. RFC 2617 (Draft Standard) (Jun 1999), http://www.ietf.org/rfc/rfc2617.txt, updated by RFC 7235
22. Fu, K., Blum, J.: Inside risks: Controlling for cybersecurity risks of medical device software. Communications of the ACM 56(10), 21–23 (Oct 2013)
23. Ghena, B., Beyer, W., Hillaker, A., Pevarnek, J., Halderman, J.A.: Green lights forever: Analyzing the security of traffic infrastructure. In: 8th USENIX Workshop on Offensive Technologies. WOOT (2014)
24. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of widespread weak keys in network devices. In: 21st USENIX Security Symposium (Aug 2012)
25. Hewlett-Packard: HP Jetdirect print servers — Using Telnet to configure the HP Jetdirect print server, http://h20564.www2.hp.com/hpsc/doc/public/display?docId=emr_na-bpj05732
26. Hewlett-Packard: HP embedded web server user guide (Aug 2007), http://h20628.www2.hp.com/km-ext/kmcsdirect/emr_na-c01151842-2.pdf
27. Hewlett-Packard: TippingPoint next-generation firewall (NGFW) technical specifications (2015), http://www8.hp.com/us/en/software-solutions/ngfw-next-generation-firewall/tech-specs.html
28. Internet Security Research Group: Let's Encrypt (2015), https://letsencrypt.org/
29. Jones, N.: Exploiting embedded devices (Jun 2012), http://pen-testing.sans.org/resources/papers/gpen/exploiting-embedded-devices-129676
30. Joyent: HTTP parser (Apr 2015), https://github.com/joyent/http-parser
31. Ketkar, C.: Standard versus proprietary security protocols. Justice League Blog (May 2014), http://www.cigital.com/justice-league-blog/2014/05/28/standard-versus-proprietary-security-protocols/
32. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd Symposium on Operating Systems Principles. pp. 207–220. SOSP (Oct 2009)
33. Kneschke, J.: Lighttpd: Fly light (Mar 2014), http://www.lighttpd.net/
34. Lafon, Y., Mendelsohn, N., Karmarkar, A., Nielsen, H.F., Hadley, M., Gudgin, M., Moreau, J.J.: SOAP version 1.2 part 2: Adjuncts (second edition). W3C recommendation (Apr 2007), http://www.w3.org/TR/soap12-part2/
35. Leroy, X., Blazy, S., Dargaye, Z., Jourdan, J.H., Tristan, J.B.: CompCert (Jun 2015), http://compcert.inria.fr/

36. Lewis, D.: Security and the Internet of Things. Forbes (Sep 2014), http://www.forbes.com/sites/davelewis/2014/09/16/security-and-the-internet-of-things/
37. Linksys: GPL code center (2014), http://support.linksys.com/en-us/gplcodecenter
38. Medin, T.: Invasion of the network snatchers: Part I. SANS Penetration Testing (May 2013), http://pen-testing.sans.org/blog/2013/05/31/invasion-of-the-network-snatchers-part-i
39. MITRE Corporation: CVE-2014-4645 (Jun 2014), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4645
40. MITRE Corporation: Common vulnerabilities and exposures (Apr 2015), https://cve.mitre.org/
41. Moore, H.D.: Penetration tester's guide to IPMI and BMCs. Rapid7Community (Jul 2013), https://community.rapid7.com/community/metasploit/blog/2013/07/02/a-penetration-testers-guide-to-ipmi
42. Nachreiner, C.: H.D. Moore unveils major UPnP security vulnerabilities. WatchGuard Security Center (Jan 2013), http://watchguardsecuritycenter.com/2013/01/31/h-d-moore-unveils-major-upnp-security-vulnerabilities/
43. Open Crypto Audit Project: Welcome to the Open Crypto Audit Project (Jun 2014), https://opencryptoaudit.org/
44. OpenSSL Project: Welcome to the OpenSSL project (2015), https://www.openssl.org/
45. OpenWRT Project: Web server configuration uHTTPd (2014), http://wiki.openwrt.org/doc/uci/uhttpd
46. Orchard, D., McCabe, F., Newcomer, E., Haas, H., Ferris, C., Booth, D., Champion, M.: Web services architecture. W3C note (Feb 2004), http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/
47. PCI Security Standards Council: Payment Card Industry (PCI) data security standard requirements and security assessment procedures version 3.1 (Apr 2015), https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-1.pdf
48. Rectanus, B.: IronBee reference manual (2014), https://www.ironbee.com/docs/manual/
49. Rocha, M., Riva, N., Falcon, F., Santamaria, P.: D-Link IP cameras multiple vulnerabilities (Apr 2013), http://www.coresecurity.com/advisories/d-link-ip-cameras-multiple-vulnerabilities
50. Rosenblatt, S.: Car hacking code released at Defcon. CNET (Aug 2013), http://www.cnet.com/news/car-hacking-code-released-at-defcon/
51. Rust Core Team: The Rust programming language, http://www.rust-lang.org/
52. Rust Core Team: Announcing Rust 1.0. Rust Programming Language Blog (May 2015), http://blog.rust-lang.org/2015/05/15/Rust-1.0.html
53. Siemens: WinCC/Web navigator: Operator control and monitoring via the web, http://w3.siemens.com/mcms/human-machine-interface/en/visualization-software/scada/wincc-options/wincc-web-navigator/pages/default.aspx
54. Spengler, B.: Grsecurity ACL documentation v1.5 (Apr 2003), https://grsecurity.net/gracldoc.htm
55. Supermicro: Supermicro intelligent management (2015), http://www.supermicro.com/products/nfo/IPMI.cfm
56. Trustwave SpiderLabs: ModSecurity: Open source web application firewall (2015), https://www.modsecurity.org/
57. Wagle, P., Cowan, C.: StackGuard: Simple stack smash protection for GCC. In: GCC Developers Summit. pp. 243–255 (May 2003)

All links were last followed on Jun 1, 2015.