Published in final edited form as:

Computer (Long Beach Calif). 2019 ; 52(12): . doi:10.1109/mc.2019.2936071.

Opaque Wrappers and Patching: Negative Results

Paul E. Black, Monika Singh

NIST, Gaithersburg, MD, USA.

Patching is a Problem

When a bug is found in software, the developer should issue a fixed version of the software. Unfortunately, when the patch is released, bad actors can analyze the changes and may be able to create an attack to exploit the bug¹. This allows a fertile attack time between when the patch is first available and when most installations are patched. The hatched area in Fig. 1 shows vulnerable machines, which have not been patched. Automated updates shorten the exposure time. However, many installations test patches before installing them to minimize breaking their processes.

A wrapper is a stopgap technique that could protect software until a fix is found. Suppose we discover a bug in software, *S*, that can be exploited by a vulnerable input, *V*. Here is a simplistic wrapper to make sure *S* is never run with *V*. (We use pseudocode for examples.)

```
if input equals V {  
    abort  
}  
run S on input
```

We can write this wrapper quickly. The problem is that an attacker can examine this wrapper and easily find the vulnerable input, *V*. The attacker can use *V* to attack machines that still run the uncorrected version.

We can use a hash function to conceal *V* from attackers. A hash function scrambles an input and produces an output that seems random. Critically, we consider hash functions where it is impractical to find an input that produces a particular output. We compute the hash of *V*, call it *V_h*, and put *that* in the wrapper, like this:

```
if hash(input) equals Vh {  
    abort  
}  
run S on input
```

¹In theory, a diversifying compiler might produce a new version whose binary and control flow is so different that simplistic comparisons do not work.

Although an attacker can get V_h from the wrapper, it is no help in finding the vulnerable input. This is the fundamental concept of an *opaque wrapper*. An opaque wrapper recognizes and blocks vulnerable inputs without leaking information about what the vulnerable inputs are.

Opaque wrappers separate protecting the software from installing a fixed version. Wrappers can be issued quickly. Users can take time to test it. Developers have time to develop a good bug fix. When enough machines are protected by the wrapper or users have had reasonable time to patch, the final bug fix can be issued. Since most machines are protected, the release of the code fix does not significantly increase the community's exposure. Fig. 2 highlights the machines that are protected because the wrapper is installed. This paper considers approaches to implementing opaque wrappers and the limitations of each.

Hash Functions for Opaque Wrappers

Unfortunately, the above wrapper with a hash function is impractical. Actual bugs have a range of vulnerable inputs, V_1 through V_n , not just one value. We could attempt to extend the above approach by hashing each input and putting them into the wrapper as follows.

```
if hash(input) equals  $V_{1h}$  OR hash(input) equals  $V_{2h}$  OR
    . . . OR hash(input) equals  $V_{nh}$  {
    abort
}
run S on input
```

For the rest of this article, we focus on the challenging part: checking the input, which we put in a function, `unsafeInput()`. The wrapper looks like this.

```
function unsafeInput(input)
{
    if hash(input) equals  $V_{1h}$  OR hash(input) equals  $V_{2h}$  OR
        . . . OR hash(input) equals  $V_{nh}$  {
        return true
    }
    return false
}
if unsafeInput(input) {
    abort
}
run S on input
```

This is still impractical. To illustrate, consider that the Heartbleed bug CVE-2014-0160 [Synopsys14] is vulnerable to messages with large payload lengths, which may be up to $65535 \cdot 2^{16}$ [Seggelmann12]. (The particulars of Heartbleed are subtle, but this is enough.)

Having thousands of compares in the wrapper is not scalable. Instead, perhaps we can use Bloom filters to quickly check.

Bloom Filters For Opaque Wrappers

A Bloom filter can confirm that an input is *not* in a set of values without revealing the set. Briefly, a Bloom filter uses a handful of different hash functions, `ha()`, `hb()`, `hc()`, ..., and a single large array of bits, `ar[]`. To add another value to a Bloom filter, `addValue()` sets the bits corresponding to all the hashes of the value:

```
function addValue(value)
{
    ar[ha(value)] := 1
    ar[hb(value)] := 1
    ar[hc(value)] := 1
    ...
}
```

To test if a value is in a Bloom filter, `testMember()` checks each bit:

```
function testMember(value)
{
    if ar[ha(value)] is 0 OR ar[hb(value)] is 0 OR
        ar[hc(value)] is 0 ... {
        return not_in_set
    }
    return probably_in_set
}
```

If any bit is zero, the value was definitely never added to the Bloom filter. To create the Bloom filter, add the value of each member of the set. It is impossible to reconstruct members of set from the bits in the Bloom filter [Naor15].

How can we use a Bloom filter to recognize vulnerable inputs? We add all the vulnerable inputs to the Bloom filter, then use the Bloom filter to implement `unsafeInput()` as follows.

```
function unsafeInput(input)
{
    Bloom_filter ar[]

    if testMember(input) equals not_in_set {
        return false
    }
}
```

```

        return true
    }

```

False positives are not a problem. First, it is easy to make Bloom filters where such false positives are vanishingly unlikely. Second, if an input is incorrectly judged to be vulnerable, then a safe input is incorrectly rejected. But Bloom filters never allow an unsafe input. Notice that an adversary cannot gain any information from the Bloom filter or the opaque wrapper that uses it.

In practice, Bloom filters are unlikely to be helpful.

Programs have large inputs, such as messages, images, or whole documents. Suppose we pass the entire input, which consists of thousands of bits, to `unsafeInput()`. To create the Bloom filter, we must add $2^{\text{thousands}}$ hashed values, which takes billions of years! In addition, the size of a Bloom filter is proportional to the number of items stored, which is larger than exabytes.

To create a reasonable Bloom filter, can we just focus on the bits of the input that are vulnerable? Let us look again at Heartbleed. The only messages that trigger the bug are heartbeat requests. Of those, messages have little risk ² if the payload length is less than, say, 60. If we write the checking function to look for heartbeat messages, then only pass the payload length to the Bloom filter, it looks like this.

```

function unsafeInput(message)
{
    Bloom filter ar[]

    if get_type(message) equals heartbeat_request {
        if isMember(get_heartbeat_payload_length(message)) {
            return true
        }
    }
    return false
}

```

The payload length is 16 bits, so only the values 61 to 65 536 are unsafe and have to be in the Bloom filter. A Bloom filter with these few thousand values takes a few seconds of computer time to create and is a few thousand bytes.

However, from this wrapper, attackers learn that the bug relates to the payload length. The attacker can try all messages with payload lengths from 1 to 65 536, discover the unsafe inputs, and construct an attack on unpatched installations. To prevent brute force attacks, we

².Any payload length longer than the payloads causes a Heartbleed buffer overflow (BOF)/Read fault. However it is unlikely that getting only, say, 61 arbitrary bytes discloses much of importance.

note a critical requirement. We must pass 60, 80, or more bits to `unsafeInput()`. Otherwise attackers can try all possible inputs.

Could we use a Bloom filter as a positive filter? That is, instead of recognizing 2^{64} unsafe inputs, perhaps there are bugs with relatively few inputs that *are* safe? This does not help. If most inputs cause failure, the software is hopelessly insecure. An attacker can just try a few random inputs and find some that exploit the software. Hereafter we will assume that the vast majority of possible inputs are safe and a relatively few cause problems.

Variations on hashes, such as a minimal perfect hash, don't help. If we can process all the inputs to construct the hash function, an adversary can try all of them.

We cannot see any way to build the `unsafeInput()` checker function using Bloom filters or other hash function techniques.

Asymmetric encryption, also known as public key cryptography, is similar to hash functions in ways. It scrambles an input to produce an seemingly random output, and the outputs are no help to find inputs that produce them. Vulnerable input values could be obscured, but the private key must be used to decrypt them before they are used. If the wrapper has the private key to decrypt vulnerable inputs, an attacker can get the wrapper and discover the vulnerable inputs, too.

In the remainder of the article, we consider Zero-Knowledge Proofs (ZKP), Fully Homomorphic Encryption (FHE), and Oblivious Transfer (OT) as building blocks to implement `unsafeInput()`. Here are the constraints that the above discussions revealed.

- `unsafeInput()` accepts a large parameter, for example, more than 60 bits. Otherwise an attacker can use a brute force attack. The parameter is an input, *I*, to be checked.
- `unsafeInput()` has the information to determine if an input, *I*, triggers a bug. This may be a list, *L*, of vulnerable values, *V*₁ through *V*_{*n*}, the minimum and maximum of ranges, or some combination.
- The attacker can reverse engineer `unsafeInput()` and extract any information that isn't irreversibly obscured, as pointed out by Michael [Michael19].
- The attacker has as much computational power to explore an exploit as the software developer has to create the patch.

Zero-Knowledge Proofs Won't Help

Briefly, in a ZKP a prover, *A*, can prove to the verifier, *B*, that the prover knows a secret value, *K*, without divulging any information about *K*³. We illustrate this in Fig. 3. A cave has two branches that connect via a locked door. *A* wants to prove to *B* that *A* know the combination to the door. *A* enters the cave and takes one of the branches. *B* stands at the intersection, picks a branch, and shouts to have *A* come from that branch. Since *A* knows the

³.Interactive and non-interactive ZKPs have very different performance, but they do not make any difference in these arguments.

combination, A can unlock the door if needed and come out of either branch. To make sure that it wasn't just luck, these steps are repeated many times. B can also make sure the door is locked. B is convinced that A knows the combination, but learns nothing about the combination. Note that A freely uses the secret value K. How can we use ZKP to implement `unsafeInput()`?

First, the set, L, of unsafe inputs is the secret value K. Second, the prover, A, uses L to decide if an input is not safe. Suppose we somehow use ZKP to implement `unsafeInput()`. In the ZKP sense, each installation is the prover, freely using the secret value, L.

But an attacker can get the wrapper, which has the list of unsafe inputs, L. Therefore, using ZKP to implement the wrapper reveals the set of vulnerable input to attackers. Hence, we cannot implement a wrapper using ZKP this way. (If we could somehow run ZKP using L but not revealing it, we could use that technique to directly implement a function to compare an input with L, but not reveal it. So ZKP does not help.)

Another approach is that the original software developer, D, is the prover. This requires every installation to communicate with D for each input. But if they are asking D to compute for each input, they can just ask D to check whether an input is in the vulnerable set. The wrapper is then trivial: it sends the input to P who returns whether it is safe or not. Again, ZKP does not help.

Fully Homomorphic Encryption (FHE) Won't Help

The notion behind FHE is that a user encrypts data, sends it to a server, which does computations the user wants without decrypting the data, and returns a result. The user decrypts the result to get the answer. The server never has access to the actual data. As a too-simple illustration, suppose the user wants to be able to get the difference between any pair of values from a set, $x_1, x_2, \dots$. The user "encrypts" them by choosing a different, very large random number for each one and sends the sums, $x_1+R_1, x_2+R_2, \dots$ to the server. For instance, x_1 is 147 and R_1 is 4 522 801. To get the difference between, say, x_3 and x_{17} , the user asks the server for the difference between the third and seventeenth sums. The server returns $(x_3+R_3) - (x_{17}+R_{17}) = x_3 + R_3 - x_{17} - R_{17}$. The user subtracts R_3 and adds R_{17} to "decrypt" the answer and recover the difference x_3-x_{17} . The server never had access to "unencrypted" values. Of course, this scheme is so simplistic that the user doesn't save time. But it illustrates the principle.

Consider implementing `unsafeInput()` using FHE. The first problem is that there is currently no range checking operation, that is, no FHE technique to determine if $\min < I < \max$. More importantly, FHE is many, many orders of magnitude slower than regular computations, which makes it impractical.

Oblivious Transfer (OT) Protocol Won't Help

In an oblivious transfer protocol, a transmitter, T, has messages $m_1, m_2, m_3, \dots$. T wants a receiver, R, to be able to get one and only one of the messages. T does not learn which one R chooses. To give the reader an intuition, T puts the messages in a box with one compartment

for each message and closes each compartment's lid. The box is built so that if any lid is opened, all other messages are incinerated. T gives R the box. R can open any lid and get just one message. This example gives an intuition of the function of an OT protocol. In practice, all OT protocols proceed by T and R exchanging encrypted commitment values.

How might we use OT to implement the checker? The software developer could fill the role of the transmitter, T. The input is an index of the messages. Each message is a bit as to whether that input is safe or not. The installation with S and the wrapper is the receiver, R. To run an OT protocol, the installation must communicate with the software developer. For an input, I, the installation gets message m_I , which is whether I is safe.

This doesn't help. As with ZKPs, if the software developer could participate, every installation can just ask the developer whether an input is safe. The only advantage is that the software developer doesn't learn the inputs that the installation is getting.

Another approach to using an OT protocol is for the wrapper to incorporate the message "boxes". The software developer transmits the message box in the wrapper. The installation can then get messages from the wrapper. Even when the attacker gets the wrapper, they can get no more than one message (whether one input is safe or not) at a time. This limits how much an attacker can learn from the wrapper.

A Few Practical Considerations

We consider briefly how to install the wrapper and where it is executed. The wrapper could be in a firewall if the inputs come via the web. The operating system might handle it, as happens with some virus checkers. Software might be built so that a wrapper can be easily added later. Also the software developer could ship a "replacement" version that just checks the input then calls S [Schaefer17].

Conclusion

An opaque wrapper would protect software from inputs that cause failures, without leaking information to adversaries to construct attacks. The function to determine if an input is not safe, the heart of an opaque wrapper, has the following requirements. 1) Its parameters must be long, say, more than 60 bits. Otherwise, an attacker can discover unsafe inputs through a brute force attack. 2) Information about which inputs are unsafe must be obscured, on the assumption that attackers can examine the wrapper. 3) It must be able to identify ranges. Encoding *each* unsafe input is impractical. 4) It cannot require communication with the software developer. Serving all installations creates a single point of failure, possible service bottlenecks, and raises concerns of leaking the installation's information to the developer.

Unfortunately, we see no practical way to implement opaque wrappers. Perhaps other techniques, such as obfuscated circuit evaluation, cryptographic accumulators, Karnaugh maps, SAT encoding, or recognition circuits will rescue opaque wrappers from being an impossible solution. Maybe advances in fully homomorphic encryption will yield an operation that returns an *unencrypted* result and is relatively fast.

Instead of trying to find one approach that works for everything, perhaps researchers can accumulate many approaches, each working in a particular situation. For instance, if only a handful of inputs trigger the vulnerability, comparing with a few hashed values works.

If the risk of attackers finding a way to exploit a bug is less than the risk of leaving a bug vulnerable, a wrapper may be useful, even if it reveals unsafe inputs.

We hope that someone figures out a way to implement opaque wrappers and make patching safer.

References

- [Synopsys14]. “The Heartbleed Bug”, <http://heartbleed.com/>
- [Seggelmann12]. Seggelmann R and Tuexen M, “RFC 6520: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension”, 2 2012 Available at <https://tools.ietf.org/html/rfc6520> Accessed 10 February 2017.
- [Naor15]. Naor Moni and Yosef Eylon, “Bloom Filters in Adversarial Environments” <https://eprint.iacr.org/2015/543.pdf>
- [Michael19]. James Bret Michael, “Assessing the Trustworthiness of Electronic Systems: The Way Forward for Reverse Engineering”, IEEE Computer, vol. 52, pp. 80–83 11 2019.
- [Schaefer17]. Schaefer Kim, private communication, 2017.

Figure 1.

An attack may be developed from a released patch. The shaded area shows machines that are vulnerable to attack because they have not been patched.

Figure 2.

The shaded area shows machines protected by a wrapper that otherwise would have been vulnerable to the attack developed from the patch.

Figure 3.

Hidden locked door analogy to Zero-Knowledge Proof. A goes to either the left or right branch at random. B comes to the fork and asks A to come from the left branch or the right branch. A can only always succeed if A knows how to unlock the door.