A Practical Approach to Protocol-Agnostic Security for Multiparty Online Services

Eric Chen (Carnegie Mellon University)
Shuo Chen (MSR Redmond)
Shaz Qadeer (MSR Redmond)
Rui Wang (MSR Redmond)

Technical Report
MSR-TR-2014-72

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

May 22nd, 2014

A Practical Approach to Protocol-Agnostic Security for Multiparty Online Services

Abstract – Serious logic vulnerabilities exist in real-world services that use security protocols such as OpenID, OAuth, and PayPal Standard. We introduce Certified Symbolic Transaction (CST), an approach for building provably secure multiparty online services. The kind of provable security that we focus on is actually an extreme form, which we call protocol-agnostic security – it holds a system implementation to an end-to-end global security predicate completely independent of the adopted protocol, and thus fundamentally guards the implementation against logic flaws in the protocol and developers' misunderstandings of the protocol.

We show that it is entirely practical for developers to apply CST on real-world systems. Unlike traditional verification approaches, CST invokes static verifications at runtime: it treats every multiparty transaction as a runtime process for creating a proof obligation for a general-purpose (thus protocol-agnostic) static program verifier. We have applied CST on five commercially deployed systems, and show that, with only tens (or 100+) of lines of code changes per party, the enhanced implementations achieve protocol-agnostic security. We also stress-tested CST by building a gambling system integrating four different services, for which there is no existing protocol to follow. Our security analysis shows that 12 out of 14 logic vulnerabilities reported in the literature will be prevented by CST. Because transactions are symbolic and cacheable, CST has near-zero amortized runtime overhead. We make the source code of these systems public, which can be immediately deployed for real-world uses.

1 Introduction

Web applications today are often multiparty systems spanning different companies, as they usually integrate third-party services like single sign-on (SSO), payment and social networking. The common practice across the industry is that developers follow the corresponding protocol documentation, such as the standards or API specifications provided by the OAuth Working Group, the OpenID Foundation, PayPal, or Facebook, to fulfill the service integration. In this paper, the word "protocol" refers specifically to a document in a natural language targeted toward software developers, although some people used it to mean generally a set of rules for distributed communication.

Protocol and security. It seems reasonable to believe that since a protocol is designed by security experts, and developers of every party try to conform to the protocol, security should be achieved. However, the belief is a well-known fallacy. Logic flaws are prevalent in real-world deployed services that adopt mainstream protocols. They defeat the whole purpose of adopting these security protocols, e.g., an attacker can purchase without paying [24][26], sign into other people's accounts without passwords [3][27], or get unintended authorizations [28].

This is largely a human miscommunication issue among developers themselves and between developers and protocol designers. For example, the Amazon Simple Pay protocol was not explicit about which party should check the consistency between the recipient of a payment notification and the payee of the payment, causing merchant websites to assume a non-existent assurance about the Amazon server [26]; developers also misunderstood the OAuth protocol, believing that any party holding an access token to read Alice's profile data can be rightfully authenticated as Alice, but did not realize that OAuth's access token represents an authorization, not a basis for authentication [28]. Even SDKs published by major companies to help adopt these protocols contain many unstated pitfalls that will subvert security [28]. The Cloud Security Alliance cites specifically this type of flaws as "Insecure Interfaces and APIs", the No.4 cloud computing top threat [8].

To solve the issue, it may appear that we would need a protocol specification to be perfect – such a high-level English document written by protocol experts would guide non-expert developers to write concrete programs for all kinds of cross-company scenarios without room for misunderstanding or misassumption. Whether this perfection is ever achievable is debatable, but the prominent protocols today are clearly far from it.

Protocol-agnostic security. In this paper, we consider it a fact of life that protocols do have room for mistakes. We introduce *Certified Symbolic Transaction (CST)*, a programming approach for developers of different parties to collaboratively achieve *protocol-agnostic security*, which holds their implementation directly to a desired end-to-end global security predicate completely independent of the protocol. The implementation is thus secured regardless of whether the protocol is foolproof or correctly obeyed by the developers. In this sense, protocol-agnostic security is an extreme form of the notion of provable security.

Traditional approaches for provable security. Provable security has been advocated by researchers for a long time, but is rarely put in practice by actual developers, especially for building real-world online services. In fact, we are unaware of any major service provider (e.g., PayPal, Facebook, Google) delivering any APIs, SDKs or frameworks toward this goal. Provable security is difficult because traditional approaches, which perform verifications *offline* (Figure 1 (upper)), essentially ask developers to prove an impossibility theorem with the verifier: no execution

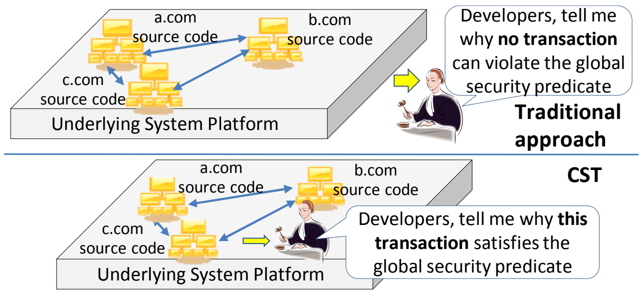

a.com
source code
b.com
source code
c.com
source code
Underlying System Platform
Developers, tell me why **no transaction** can violate the global security predicate
Traditional approach
a.com
source code
b.com
source code
CST
c.com
source code
Underlying System Platform
Developers, tell me why **this transaction** satisfies the global security predicate

Figure 1: CST imposes a much less demanding obligation

(not even the most creative, bizarre and unrealistic ones) can violate the security predicate. It requires developers' adversarial and inductive thinking about an unbounded number of all executions that the verifier considers possible. This is especially hard at the source code level for multiple parties. Furthermore, offline verification requires developers to model the underlying platform, which is expertise-demanding and subjective.

The CST approach. CST achieves the same provable security goal in a different a manner, shown in Figure 1 (lower). It treats every protocol execution (referred to as a *multiparty transaction* from now on) as *a runtime process for creating a proof obligation* for a general-purpose (thus protocol-agnostic) static program verifier, which we call *the certifier*. The certifier logically examines whether the computations on all parties in this transaction collectively ensure the global predicate.

Compared to traditional approaches for provable security, the runtime nature of the certification in CST demands much less from developers. Since transactions failing to be certified will be rejected, developers are only required to make sure that every intended transaction is logically sound. Thus, a developer performs constructive thinking about the expected, not adversarial thinking about the unexpected. Moreover, modeling the underlying platform is unnecessary, since every execution is generated by the actual platform.

The most novel aspect of the CST approach is the use of static program verification at runtime. In Section 2, we motivate why static verification is essential for representing and certifying multiparty transactions. Although program verification is prohibitively expensive in general, CST combines static verification and caching to ensure near-zero amortized overhead.

Real-world demonstration. Our main contribution is to show that it is entirely practical for actual developers to apply CST on real-world systems. We have applied CST on five commercially deployed systems, and show that, with only tens (or 100+) of lines of code changes per party, the original implementations are enhanced to achieve protocol-agnostic security. The systems are based on real-world frameworks, such as the PayPal and Amazon Payments services, Microsoft LiveID single-sign-on SDK, the OpenID framework in DotNetOpenAuth, the OAuth template in Visual Studio ASP.NET MVC 4, and the NopCommerce software. We also stress-tested CST by building a gambling system integrating four services, for which there is no existing protocol to follow. All the systems and their source code are accessible from our anonymous project page at https://sites.google.com/site/protocolagnostic/. We conducted evaluations on three aspects:

- *Security*. We analyzed real vulnerabilities reported in the literature that we are aware of, which includes 14 cases. CST will foil the exploits or avoid the vulnerabilities in 12 out of the 14 cases, the other two being out-of-scope issues.
- *Protocol-agnosticism*. We show that considerably different protocols, e.g., Amazon Simple Pay vs. PayPal Standard for payment and OpenID vs. OAuth vs. Live Connect for single-sign-on, can be held to the same global predicates. More interestingly, we show several implementations that blatantly violate the OAuth 2.0 protocol, but satisfy the end-to-end global predicate for single-sign-on. They all turn out to be as secure as the protocol-conformant ones.
- *Performance*. Because symbolic transactions are cacheable, CST incurs only near-zero amortized runtime overhead, thus is suited for real-world uses.

2 Overview of CST

To understand the technical question that CST needs to answer, we first need to recognize two basic characteristics about typical multiparty online services:

(1) *No global data storage*. Each party holds its own data structures (e.g., databases) without a globally shared storage, because different parties are distrustful services despite their cooperation on specific transactions. For example, PayPal's payment record database should not be exposed to any merchant website, because it contains other merchants' transactions. Similarly, Facebook's single-sign-on service should not expose its data structures to any relying party website, because it contains many secret strings and IDs of users of other relying parties.

(2) *Security as a global property*. Security is a property across different parties. For example, "secure checkout" requires that an order on the merchant website has a corresponding payment record on the PayPal server. For this reason, the security predicate to be examined by CST needs to reference data structures of different parties. We call it an *ambient predicate*.

Figure 2 shows a simplified system that preserves the essence of a realistic system (to be described in Section

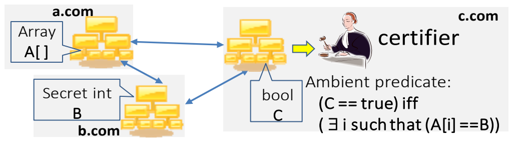


Figure 2: A simple multiparty system

3). Data structures A[], B and C are held by three companies. They try to jointly achieve the ambient predicate (C == true) iff (∃ i such that (A[i] ==B)), but their implementation (or even the protocol they adopt) may be logically flawed. The technical question for CST is: *how can the certifier (in this case, on c.com) check the ambient predicate, given that it has no access to the array A[] or secret integer B of the other parties*.

Solution. Obviously, the ambient predicate cannot be concretely checked, because A[] and B are not shared with *c.com*. The alternative approach, which is the core idea of CST, is to check it symbolically, i.e., to examine whether the sequence of computations of a multiparty transaction *logically* implies the ambient predicate.

CST achieves this using a message field called *SymT* (i.e., Symbolic Transaction), which is attached to every message to collect the source code executed on each party. In general, disclosing source code to another party may lead to intellectual property infringement or reverse-engineering. For CST, this is not a serious concern, because the disclosed source code just implements the protocol that every party has agreed on.

When a transaction completes, the certifier uses the final SymT value to synthesize a program representing the executed source code of the entire multiparty transaction. Symbolic verification then checks that the program satisfies the ambient predicate. How SymT collects the source code and how the certifier synthesizes the program are the focal points of CST, which we will elaborate in Sections 3 and 4.

The complexity of verifying a symbolic transaction depends on the expressiveness of the ambient predicate and the program fragments executed by the different parties. We found first-order logic to be adequate for the services we studied and use an off-the-shelf automated program verifier based on satisfiability-modulo-theories [21] for the verification.

In general, symbolic verification is expensive. However, for CST, it incurs an extremely low amortized cost, even lower than what a concrete checking would incur (which would need network messages). The certifier achieves this efficiency by caching the theorem proved by it about a symbolic transaction. Since the theorem holds for all inputs, a future identical symbolic transaction is deemed convincing immediately regard-less of the data values on which it computes. If the source code is unchanged, this caching results in near-zero amortized runtime overhead. Furthermore, the caching is over all transactions generated by all users.

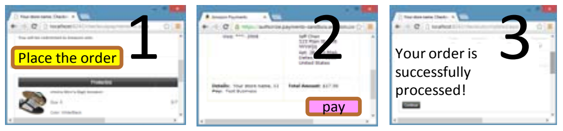


Figure 3: The basic checkout/payment flow.

Most likely, developers themselves are the users who trigger the verification, and real users enjoy the caching.

Threat model. We assume that the attacker has browsers and his/her own servers, but does not control the servers of non-attacker parties. Developers of non-attacker parties are cooperative, and do not lie about the executed source code. The network traffic is protected by HTTPS, so the attacker cannot read or tamper with data in transit. We assume that all implementations are free of general programming bugs such as buffer overrun, cross-site scripting and cross-site request forgery, which are orthogonal to the type of flaws that CST targets.

3 An Illustrative Example

We now give an example about secure checkout to explain the CST approach. Figure 3 shows the basic steps in every checkout transaction (the large buttons and text are added for clarity): (1) placing an order on the merchant site; (2) making a payment on the cashier site; (3) completing the order on the merchant site.

Figure 4 shows the three parties in the transaction – the client and two servers. We assume that *Cashier.com* is the cashier site, and *TStore.com* is the merchant site. The client is a greedy shopper who wants to check out without making a full payment or any payment at all. As the adversary, the client's behavior is arbitrary. Essentially, the "client" can be thought of as the wild Internet that can send arbitrary HTTP requests in any arbitrary order, not conforming to Figure 4.

There are three web methods. Each is invoked by an HTTP request, and returns an HTTP response. Without loss of generality, we name them placeOrder, pay and completeOrder. Accordingly, requests and responses are named placeOrder_req, pay_resp and so on.

Data structures. Every transaction involves the data structures on the two servers: orders[] is an array to store all orders, indexed by the identifier of each order (i.e., orderID); mySellerID is the merchant's identifier registered on the cashier; payments[] is the payment records on the cashier. A real implementation may use database tables instead of arrays. Section 5.3 will explain how we convert database operations into array accesses by defining "stub methods".

Security predicate. The security predicate ranges over the multiparty transaction. Section 4.2 will give a thorough explanation about the SymT representation.

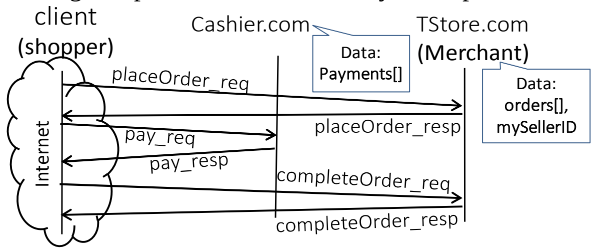


Figure 4: The basic messages and data for checkout

For now, let's assume the transaction is well represented, and all the messages are with respect to this transaction. Below is a predicate defining secure checkout (the line numbers are added for easy reference).

```
placeOrder_req.orderID == completeOrder_req.orderID &&  (1)
∃ i such that (                                          (2)
  Cashier.payments[i].status == "Paid" &&                (3)
  Cashier.payments[i].total
        ==TStore.orders[placeOrder_req.orderID].gross && (4)
  Cashier.payments[i].payee == TStore.mySellerID &&      (5)
  Cashier.payments[i].orderID == placeOrder_req.orderID) (6)
```

Line 1 ensures that the order ID in the request for completeOrder is still the same as the one in the initial request for placeOrder. Lines 2-6 assert that there exists a payment on the cashier which satisfies the following four conditions: the payment's status is "Paid" (line 3); the payment total matches the order gross (line 4); the payment is made to the correct account (line 5); and the payment is for the correct order (line 6).

Note that the discussion up to this point is not specific to any protocol, but only our *definition of the problem*.

3.1 A traditional implementation

The predicate above specifies the security objective. However, it is not locally checkable because it is about data relations across different parties. For example, payments[] is the cashier's data structure, while orders[] is the merchant's. Therefore, it is an ambient predicate.

Protocol specifications today do not explicitly define their ambient predicates. Instead, a protocol simply instructs each party how to check a set of locally checkable predicates and respond to other parties. It is *hoped* that security is achieved as a result of all these local checks. As mentioned in the introduction, this is a fallacy in reality.

Figure 5 shows a simplified example of a traditional implementation. It defines the data structures explained earlier, and implements placeOrder(), completeOrder() and pay() to handle *https://TStore.com/placeOrder.aspx*, *https://TStore.com/completeOrder.aspx* and *https://Cashier.com/pay.aspx*. Let's assume the client is checking out a $35 order with orderID 123. In a non-malicious scenario, the messages are as follows (readers can walk through the code in Figure 5 to understand how the messages are generated). For brevity, every message is represented by enclosing data fields in angle brackets after the message name, e.g., the first message is *https://TStore.com/placeOrder.aspx?orderID=123.*

(1) placeOrder_req:
placeOrder_req<orderID=123>

(2) placeOrder_resp and pay_req (a browser redirection):
pay_req<orderID=123,total=35,returnURL=https://TStore.com/completeOrder.aspx,signature=[TStore's signature for the whole request]>

(3) pay_resp and completeOrder_req (a browser redirection):
completeOrder_req<orderID=123,status=Paid,signature=[Cashier's signature for the whole request]>

```
class Merchant  {
   order_record_t[] orders;
   string mySellerID = "JohnSmith1234";
   PlaceOrderResp_PayReq  placeOrder (PlaceOrderReq req)  {
      PlaceOrderResp_PayReq resp;
      int orderID = req.orderID;   resp.orderID = orderID;
      orders[orderID].status = "Pending";
      resp.redirectionURL = "https://Cashier.com/pay.aspx";
      resp.total = orders[orderID].gross;
      resp.returnURL = "https://TStore.com/completeOrder.aspx";
      sign(resp); return resp;
   }
   public bool completeOrder(PayResp_CompleteOrderReq req){
      if (VerifySignature(req)==false) return null;
      if (req.signer != "Cashier.com" ||  req.status != "Paid" ||
            orders[req.orderID].status != "Pending")  return false;
L1:   orders[req.orderID].status = "Complete"; return true;   }
}
class Cashier  {
  payment_record_t[] payments;
  PayResp_CompleteOrderReq pay(PlaceOrderResp_PayReq req){
      if (VerifySignature(req)==false) return null;
      i=getAvailableIndex();
      payments[i].payee = req.signer;
      payments[i].orderID = req.orderID;
      payments[i].total = req.total;
      PayResp_CompleteOrderReq resp;
      resp.redirectionURL = req.returnURL;
      resp.orderID = req.orderID;  resp.status = "Paid";
      sign(resp);  return resp;  }
}
```

Figure 5: A traditional implementation.

Assuming that signing and signature checking are done correctly, readers can confirm that the message sequence above can drive the code to Line L1, where the order is marked *Complete*. However, there is a problem: *the ambient predicate we care about is nowhere to be found in Figure 5*. The developers' hope is that the local checks in these methods have collectively ensured "security". Is it really so?

A real-world vulnerability. In fact, this example is based on the real Amazon Simple Pay payment method. An exploitable logic flaw was detailed in Section III.A.2 of reference [26]. In the exploit, the attacker has his own seller account *Mark* and server *MarkStore.com*, and is able to purchase from the victim *TStore* by only paying to *MarkStore*. The details are: when he receives placeOrder_resp from *TStore*, he discards the signature and resigns it as *MarkStore*. This message is sent to the cashier (Amazon) as pay_req. From the cashier's point of view, the situation is that the attacker is purchasing an order from *MarkStore*, so Mark gets paid. However, the redirectURL points to *TStore.com*, so TStore receives the completeOrder_req signed by the cashier. TStore does not expect the cashier to notify it about an irrelevant payment (i.e., a payment made to Mark), and is fooled to complete the order. NopCommerce [22], a popular e-commerce software, is subject to this flaw.

3.2 The CST-enhanced implementation

CST enhances the implementation by requiring a SymT field in each message, which contains SHA-1 hash values of the source code of invoked methods. For example, the source code hash of placeOrder is f8f8bd5b0fe4711a09731f08c06c3749d240580c. For *readability of this paper*, we show a hash value as a hash symbol "#" with a method name, e.g., #placeOrder, but *a real SymT does not contain "#" or method names.*

SymT is now attached to every message shown earlier (parentheses and colons to be explained in Section 4.1):

(1) placeOrder_req:
placeOrder_req<orderID=123,SymT=empty-string>

(2) placeOrder_resp and pay_req (a browser redirection):
pay_req<orderID=123,total=35,returnURL=https://TStore.com/completeOrder.aspx,SymT=TStore.com::#placeOrder(), signature=[TStore's signature for the whole request]>

(3) pay_resp and completeOrder_req (a browser redirection):
completeOrder_req<orderID=123,status=Paid,SymT=Cashier.com::#pay(TStore.com::#placeOrder()),signature=[Cashier's signature for the whole request]>

At the end, completeOrder_req invokes completeOrder(…) (see Figure 6). The final SymT "TStore.com::#completeOrder(Cashier.com::#pay(TStore.com::#placeOrder()))" is given to the certifier (which runs on *TStore.com*). Based on the SymT, the certifier synthesizes a program, which we call *vProgram*. We defer the synthesis algorithm to Section 4.2. The vProgram is formally verified against the ambient predicate. If the verification is passed, the transaction is approved (e.g., the order is marked *Complete*). Otherwise, it is rejected.

Due to the space constraint, we show the vProgram in Appendix A. Lines L4 and L5 specify the ambient predicate. This vProgram can be verified, suggesting that the corresponding transaction is secure.

The synthesis of the vProgram requires the certifier to recover source code texts from their hash values. This capability relies on a "de-hash" table, which contains hash-to-source-code mappings. Anyone can submit a source code text to the de-hash table so that its hash value will be computed and associated with it.

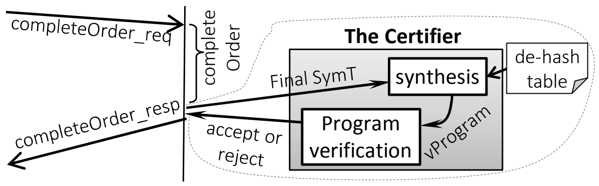


Figure 6: The certifier module on TStore.com.

A preview of security. If the logic flaw explained earlier is exploited, the final SymT will be either TStore.com::#completeOrder(Cashier.com::#pay()) or TStore.com::#completeOrder(Cashier.com::#pay(MarkStore.com::#placeOrder())), indicating that placeOrder is either not performed or performed by *MarkStore*. The two SymT values are actually equivalent, since a computation on *MarkStore* does not mean anything to *TStore,* who only trusts itself and *Cashier.com*. In the next section, we will see that the vProgram synthesized from either SymT will have Line L3 replace Lines L1 and L2 in Appendix A. The havoc operation in Line L3 makes pay_req unconstrained, and fails the verification. The transaction is thus rejected.

4 The CST Approach

This section elaborates CST more comprehensively.

4.1 The certifier

The certifier is a Boolean method:

```
bool certify(string FinalSymT, string AmbientPredicate,
                 string[] TrustedParties)
```

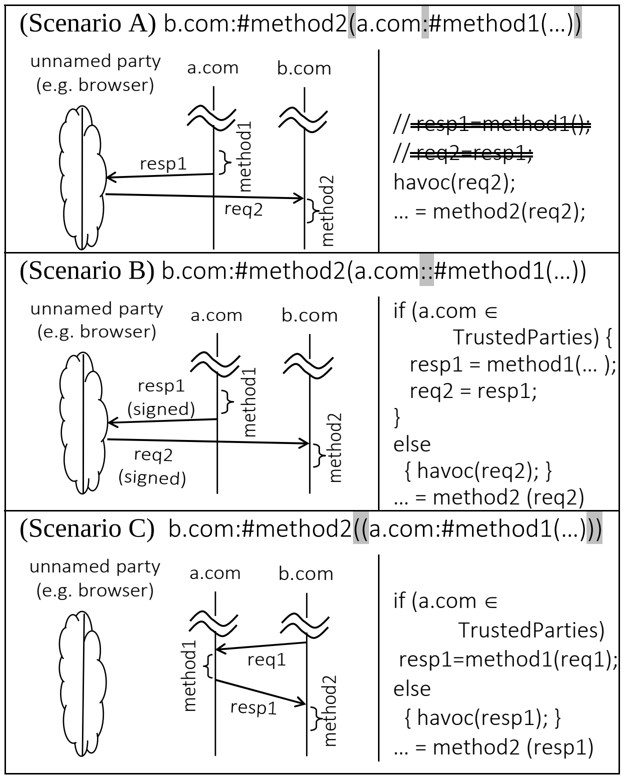
The arguments FinalSymT and AmbientPredicate are self-explanatory. The argument TrustedParties is an array to specify which parties are considered trusted for this ambient predicate. For the example discussed earlier, the certifier (on *TStore.com*) only needs to trust *TStore.com* and *Cashier.com*, i.e., the validity of the ambient predicate should not depend on any other party. Similarly, in the single-sign-on scenario, the certifier on the relying party *foo.com* should only trust *foo.com* and the identity provider (e.g., *facebook.com*), but no one else. As stated earlier, *the client (browser) is always an untrusted party, involved in all transactions*. TrustedParties decides which computation steps the certifier should take into account. Data from other parties, including the client, are "havocked" in the synthesized vProgram, which we will explain shortly.

4.2 SymT (symbolic transaction)

SymT is the basis of the CST approach. It makes multiparty transactions explicitly represented in protocol messages. This contrasts to today's web programming, in which only sessions (pairwise between two parties) are explicit for developers, while multiparty transaction is largely a notion in protocol designers' minds.

SymT needs to record not only the sequence of method calls, but also how two consecutive calls are stitched, i.e., how the output of a call (referred to as method1 on *a.com*) is fed into the input of the next call (referred to as method2 on *b.com*). Specifically, the main question is why *b.com* should believe that the input of method2 indeed comes from *a.com*. There are only two possible reasons: (1) the input is signed by *a.com*; (2) *b.com* itself makes a direct server-to-server call to method1 to obtain the input for method2.

Therefore, SymT must precisely encode the stitching scenarios. Figure 7 shows three SymT values, in which we highlight certain symbols for discussion. In scenario A, the output of method1 is not signed (denoted by the highlighted single-colon), and is supplied to method2 by an unnamed party (denoted by the highlighted parentheses). An unsigned browser redirection is a typical scenario of this. The only difference in scenario B is that the input of method2 (i.e., the output of


(Scenario A) b.com:#method2(a.com:#method1(…))
unnamed party (e.g. browser)
a.com
b.com
resp1
req2
method1
method2
(Scenario B) b.com:#method2(a.com::#method1(…))
resp1 (signed)
req2 (signed)
(Scenario C) b.com:#method2((a.com:#method1(…)))
req1
resp1

```
//~~resp1=method1();~~
//~~req2=resp1;~~
havoc(req2);
… = method2(req2);
```

```
if (a.com ∈
        TrustedParties) {
  resp1 = method1(… );
  req2 = resp1;
}
else
  { havoc(req2); }
… = method2 (req2)
```

```
if (a.com ∈
        TrustedParties)
 resp1=method1(req1);
else
  { havoc(resp1); }
… = method2 (resp1)
```

Figure 7: Stitching two method calls

method1) is protected by *a.com*'s signature, so *b.com* is confident that it is generated by *a.com*, untampered. The signing is denoted by the double-colon "::". In scenario C, method1 is called from *b.com* using a direct server-to-server call (e.g., a SOAP or REST API call), so *b.com* of course has the confidence that the input of method2 comes from *a.com*. The server-to-server call is denoted by the pair of double-parentheses "((…))".

Stitching in the vProgram. An important operation used in the vProgram is *havoc(x)*, which is to assign variable x a non-deterministic symbolic value, and thus expresses the notion that the vProgram distrusts the variable. The right column in Figure 7 gives the vProgram snippet for each SymT. In scenario A, the highlighted single-parentheses and single-colon indicate that an unnamed party (e.g., browser) supplies an unsigned req2 to method2, so req2 should be assumed arbitrary. This arbitrariness is specified as havoc(req2). Because of the havoc, calling method1 has no effect on method2. For the presentation purpose, we show the two statements crossed out. Essentially, any method invoked by an unsigned browser redirection should assume nothing about its prior computations.

In scenario B, the double-colon indicates that resp1/req2 is signed by *a.com*, so even though it is still a browser redirection, we should not havoc req2 if *a.com* is a trusted party. Scenario C is a server-to-server call. Whether to havoc resp1 only depends on whether *a.com* is a trusted party (Note: the trustworthiness of *b.com* does not matter in this stitching step. It does affect how method2 will be stitched later).

Another valid SymT, not shown in Figure 7, could be b.com:#method2((a.com::#method1(…))), representing a server-to-server call returning a signed response. It is equivalent to scenario C, as the signing is unnecessary.

Summary. SymT is generalized by the production rules in Table 1. The rules we actually use, given in Appendix B, are slightly more comprehensive, which accommodate multiple arguments in a method call and a message with selective fields covered by a signature.

Table 1: Production rules for SymT

SymT	→	*empty*
	|	PARTY-ID *:* METHOD-CALL
	|	PARTY-ID *::* METHOD-CALL
METHOD-CALL	→	SRC-HASH *(* SymT *)*
	|	SRC-HASH *((* SymT *))*
PARTY-ID	→	*a.com* | *b.com* | *amazon.com* | …

4.3 Signature checking

CST assumes that all the signatures are correctly checked at runtime: suppose a signed message carries the SymT a.com::#f(…), the receiver must ensure that the signature is generated by *a.com*. Vulnerabilities due to signature checking are out of scope of CST. It is also worth noting that: (1) CST incurs no additional signing operation, i.e., any unsigned message in the original implementation will remain unsigned in the CST-enhanced implementation; (2) For any signed message, *the SymT field itself must be covered by the signature.*

4.4 Verification tool chain

We demonstrated CST on systems implemented using ASP.NET and C#. The focus on .NET is because of the tool chain that we use for verification, as opposed to any fundamental limitation of CST. Figure 8 shows the tool chain. A vProgram is compiled by the C# compiler of Visual Studio. The output is an executable file of .NET byte code. ByteCodeTranslator (BCT) [4] is a tool to translate a .NET byte code program into a Boogie program. Boogie is an intermediate verification language [7]. We use the Corral system [14] as the verifier. In addition to the input Boogie program, the Corral verifier expects a non-negative number to establish a bound for the unfolding of loops and recursion in the program. Corral outputs exactly one of three results: the program is verified, or the program is verified with respect to the bound, or the program is falsified. In the final case, Corral also presents a counterexample witnessing the error in the program. The certifier certifies the transaction presented to it *only if Corral returns the first output.*

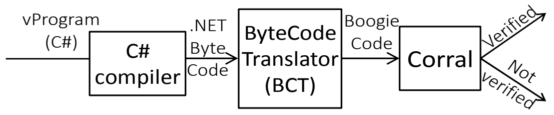


Figure 8: Our tool chain

An ambient predicate is specified using the Contract class [19] defined in C# System.Diagnostics.Contracts namespace. Lines L4 and L5 in Appendix A, discussed earlier, use methods Contract.Assert and Contract.Exists.

4.5 Caching the certification results

Program verification is expensive (e.g., 10 - 30 seconds to verify a transaction in our cases). It is impractical to do it for every transaction. Caching is therefore essential in CST: when the verification is done, the certifier caches the result (i.e., true/false) with the input arguments FinalSymT, AmbientPredicate and TrustedParties. Any future call to the certifier by any user will return the result directly if it hits the cache.

5 Applying CST in the Real World

We have applied CST to enhance various systems that serve practical purposes. Unlike proof-of-concept prototypes, these systems contain realistic source code and data structures written by actual developers. We view it as an accomplishment that all our enhanced systems are ready for commercial deployments. For example, people can install our CST-enhanced NopCommerce to run their stores: a customer can choose items, check out orders, and specify shipping and payment methods, etc; payments are made on the real Amazon and PayPal servers. People can also use our Live Connect SDK to enable single-sign-on on their websites. Functionality-wise and performance-wise, our systems are indistinguishable from the original ones, except that the implementations are provably secure. These systems are all publicly accessible. Their URLs, source code, instructions and videos are given in https://sites.google.com/site/protocolagnostic/ [31].

5.1 Categories of the enhanced systems

We have worked on three categories of systems so far:

Payment/checkout. NopCommerce [22] is a widely used open-source e-commerce application. It was one of the focused systems in previous security studies [26][29]. NopCommerce accepts many third-party payments. We decided to enhance its payment modules for Amazon Simple Pay and PayPal Standard. Their message diagrams are given in Appendix C1. These two payment protocols are significantly different in that Amazon Simple Pay is based on signed redirection messages (which we show in Section 3.1), whereas the PayPal Standard mechanism relies on a direct server-to-server call, namely the PDT (Payment Data Transfer) query, for securely communicating the payment details.

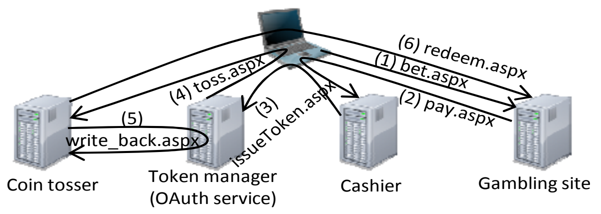


Figure 9: The gambling system.

Single-sign-on (SSO). We worked on the implementations of three different SSO protocols: (i) the *OpenID-2.0*-based SSO [23] in the DotNetOpenAuth framework [10]; (ii) the *OAuth-2.0*-based SSO [13] in Microsoft Visual Studio ASP.NET MVC 4 web application template that uses Facebook's OAuth service [11]; (iii) Live Connect SDK [18], which heavily influenced the *OpenID Connect* specification [25]. (Note that the terminology may cause a little confusion. *OpenID Connect* is a protocol, drafted by the OpenID Foundation, to use OAuth 2.0 for SSO. It was published very recently. Live Connect SDK predates the OpenID Connect specification, so the SDK refers to its SSO mechanism as OAuth 2.0, rather than OpenID Connect.) The message diagrams are shown in Appendix C2.

Gambling. People are familiar with the above two categories, because standards organizations and major companies have provided protocol specifications or API documentations. We decided to use CST to build a gambling system. The goal is two-fold: (1) we do not have any existing gambling protocol to conform to, so building this system is an end-to-end exercise of the protocol-agnostic thinking process; (2) previous scenarios only involve two trusted services. We want to challenge the CST approach by involving more parties. The gambling system we built consists of four independent services for betting, payment, authorization and coin-tossing (see Figure 9).

5.2 Ambient predicates

Despite the significant differences among these systems and their adopted protocols, we check the same ambient predicate for each category.

Payment/checkout. Our enhanced implementations for Amazon Simple Pay and PayPal Standard ensure the same ambient predicate as we presented in the example in Section 3. It is to ensure that when the merchant is about to check out an order, there exists a payment record in the cashier that matches this order.

SSO. Figure 10 gives our protocol-agnostic definition of the SSO security goal, regardless of whether a system follows OAuth 2.0, OpenID 2.0, or Live Connect. In every SSO system, there are a client, an identity provider (IdP) and a relying party (RP). The client holds a piece of ClientSecret, which is shared with the identity provider, but not the relying party. The relying party has at least two pieces of data: My_Realm is its identifier known to the identity provider; My_Hostname is its network-addressable name.

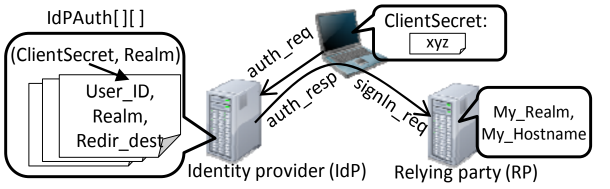


Figure 10: A protocol-agnostic definition of single-sign-on

An SSO transaction starts with a request from the client to the identity provider, namely auth_req, containing the ClientSecret and the Realm of the relying party that the client wants to sign in. The identity provider then retrieves an object called ID_Claim using the pair (ClientSecret, Realm). ID_Claim contains at least three fields: User_ID is the identifier of the user that this claim is about; Realm is as described above; Redir_dest indicates the destination of the redirection message (i.e., auth_resp followed by signIn_req in the figure). The retrieval is based on a two-key dictionary called IdpAuth, which is defined as follows in C#:

```
Dictionary<string, Dictionary<string, ID_Claim>> IdpAuth;
```

Note that how IdpAuth entries are established is not what SSO concerns about. The identity provider can identify the client as "Alice" for any reason (e.g., through password or SSL client certificate), thus create an entry. An SSO protocol is to prove to the relying party the existence of the entry, i.e., the fact that the identity provider somehow believes the client is Alice.

Assuming final_result is the variable that the relying party uses to represent the User_ID of the final authentication result, the ambient predicate is:

```
IdPAuth[auth_req.ClientSecret][auth_req.Realm].Realm
                              == My_Realm      &&          (1)
IdPAuth[auth_req.ClientSecret][auth_req.Realm].Redir_dest
                              == My_Hostname   &&          (2)
IdPAuth[auth_req.ClientSecret][auth_req.Realm].User_ID
                              == final_result              (3)
```

The first clause asserts that the ID_Claim is generated for signing into *this* relying party (i.e., the one that the certifier resides on). The second clause asserts that the identity provider passes the ID_Claim to *this* relying party, not to any other website (which could then use the ID_Claim to sign into *this* relying party illegally). The third clause asserts that the final authentication result the relying party takes is the one in the ID_Claim.

Gambling. The ambient predicate for the gambling system is given below. Clauses (2)-(4) ensures that a proper payment has been made for the bet (identified by final_req.betID); Clauses (5) - (8) ensures that the bet is valid and matches the tossing result of the coin-tosser. The validity of the predicate depends on the computations on all four services.

```
GamblingSite.bets[final_req.betID].status=="Pending" &&          (1)
∃ i such that (
   Cashier.payments[i].total ==
         GamblingSite.bets[final_req.betID].amount   &&          (2)
   Cashier.payments[i].orderID == final_req.betID   &&           (3)
   Cashier.payments[i].payee==GamblingSite.MySellerID)   &&      (4)
∃ x such that (
   TokenMgr.records[x].payee==GamblingSite.MySellerID && (5)
   TokenMgr.records[x].betID == final_req.betID   &&             (6)
   TokenMgr.records[x].EffectiveResult != "untossed" &&          (7)
   GamblingSite.bets[final_req.betID].guess
         == TokenMgr.records[x].EffectiveResult)                 (8)
```

As mentioned earlier, a motivation for building this gambling system is to challenge the CST approach with substantial complexity. In this case, the final SymT of a normal transaction contains 4 parties and 7 hash values:

```
GamblingSite.com:#redeem(CoinTosser.com::#post_to
ss((TokenMgr.com:#write_back(CoinTosser.com:#toss(
TokenMgr.com::#issueToken(amazon.com::#pay(Gamb
lingSite.com::#bet()))))))))
```

The synthesized vProgram has more than 300 lines of C# code, which the certifier is able to verify.

5.3 Programming effort

Every verification technology applied to real-world systems needs the efforts of abstraction – to draw a line between the levels to be verified and the levels of which the details are assumed or ignored, because it is impossible to verify the entire system at every level. The abstraction is often done through *factoring* and *stubbing*. This subsection explains what they mean in the CST programming. For concreteness, the description incorporates our experience of enhancing the Live Connect SDK, although the general ideas apply to our experiences of enhancing other systems.

(1) Factoring. CST requires the core computations to be factored out in order to be logically verified. Typical non-core computations include methods for parsing, composing and HTTP-encoding/decoding for messages. These non-core computations contain complicated string (byte-array) operations. Currently Corral and Boogie have only limited capability for reasoning about string operations: assignment, equality comparison and string-indexed dictionary are supported, but concatenation, tokenization, character operations, etc. are not.

Figure 11 shows the call-graph of callback.aspx in the Live Connect SDK, which handles a redirection from the LiveID server. In this 3-dimensional drawing, the methods are placed on an unshaded level and a shaded level. The shaded level consists of the core methods that we factor out. The unshaded level serves as the interface between the core logic and the underlying platform. For example, Page_Load parses HTTP arguments. RequestAccessToken_raw is a method we create so that the string operation for constructing an HTTP request can be separate from the core method RequestAccessToken. The constructor of class JWT (i.e., JsonWebToken) performs Base64 decoding and signature validation, which are byte-array operations.

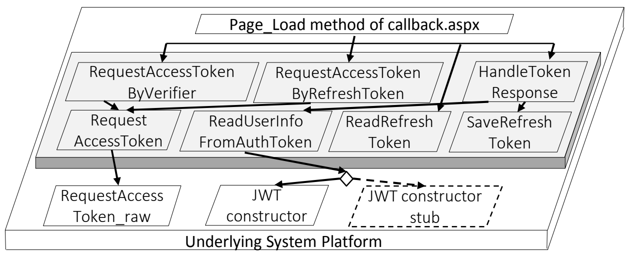

Page_Load method of callback.aspx
RequestAccessToken ByVerifier
RequestAccessToken ByRefreshToken
HandleToken Response
Request AccessToken
ReadUserInfo FromAuthToken
ReadRefresh Token
SaveRefresh Token
RequestAccess Token_raw
JWT constructor
JWT constructor stub
Underlying System Platform

Figure 11: Factoring and stubbing in callback.aspx

It is not a requirement that all complicated string (byte-array) operations are moved out from the core methods. For example, a core method can still construct a string for the logging/debugging purpose, as long as it does not affect the validity of the ambient predicate. Our experience on existing implementations of real-world service frameworks is that they are already architected similarly to Figure 11 so that lower-level methods parse HTTP requests into well-structured objects and assemble HTTP responses using these objects, while upper-level modules implement core computations on these objects. The core computations usually deal with basic types (e.g., integers and Booleans), structs and arrays of basic types, as well as string assignments and equality comparisons. Corral/Boogie can effectively reason about all these programming constructs.

(2) Stubbing. The core methods call many other methods, which will not be included in the vProgram for verification. In other words, these methods are treated as unimplemented, from the certifier's standpoint. For Corral/Boogie, the default semantics of an unimplemented method is that it returns a non-deterministic value, but does not modify any program state (i.e., the body is a no-op). This works in most cases, because most of these methods are not essential to the verification. However, there are a few situations in which the semantics of these methods matter, so developers need to define their semantics as stubs. In the Live Connect SDK, we provided a stub method as shown in Figure 11. The source code is below.

```
static JsonWebToken JWT_Constructor_stub(OAuthToken token)
{     JsonWebToken jwt;
      havoc(jwt);
      Contract.Assume(jwt == token.jwt && jwt != null);
      return jwt;
}
```

The reason to provide JWT_Constructor_stub is to replace the Base64 decoding and signature validation operations in the JWT constructor with the logic most essential to the verification. Specifically, the logic is that a new JWT object equals to the jwt member of the input argument token, i.e., jwt!=null && jwt== token.jwt.

Another situation for providing a stub is to model a database operation. In real-world systems, persistent data (e.g., the payment records) are often stored in and queried from a database by INSERT and SELECT. Corral/Boogie does not have built-in support for these operations. Developers need to wrap these operations in C# methods, and define stubs that are logically equivalent to database operations but use C# data structures like array, set, list, etc.

5.4 Caveats in the current implementations

The ambient predicates defined above use generic names. Unfortunately, different protocol committees and service companies have not made the effort to consolidate them, so currently the same concepts are named differently across protocols and implementations. For example, the ClientSecret in our definition is called MSPAuth in Live Connect; the Realm in our definition is called AppID in Facebook OAuth and openid.realm in OpenID 2.0. Therefore, the same ambient predicates need to adapt to the different sets of names. This does not mean that the predicates are protocol dependent, but is only due to the unconsolidated terminologies.

We do not have access to source code of the real Amazon, PayPal, Microsoft or Facebook services. For each of these services, we built a "wrapper service", which serves as a relay that adds source code hashes to SymT fields. We used our best effort to provide C# methods to approximate the behaviors of the actual APIs. We wish that, in the future, these companies can attach the actual source code hashes for these APIs.

5.5 Lines of code that we added or changed

Appendix D shows the lines of code (LoC) we added or changed in each project, excluding comment and white lines. The certifier is the same across all projects. It consists of 347 LoC. The LoC numbers in the unshaded cells are a good measurement of the effort for factoring and stubbing. They are fairly small, under 200 LoC for each party, indicating that the original developers had architected the code well so that it was amenable for the CST enhancement. The shaded cells correspond to services that we had to build by ourselves, so factoring and stubbing do not apply. These include wrapper services and the four services for gambling.

6 Evaluation

We evaluated the CST approach on security, protocol-agnosticism and performance.

6.1 Security

We studied 14 real-world vulnerabilities (see Appendix E), which we believe are a representative sample set in this problem space – this set includes all the cases reported in the literature [24][26][27][28] that allows an attacker to either check out an order without a proper payment or sign into a victim user's account through SSO, excluding the cases due to generic web programming flaws like cross-site scripting, cross-site request forgery (CSRF) and session fixation. We show next that 12 out of the 14 cases would be addressed by CST. The remaining two are out-of-scope issues.

Cases for which CST is effective. Attacks for cases #1, #2 and #11 can be launched against the systems that we built using CST – NopCommerce with PayPal Standard, NopCommerce with Amazon Simple Pay and OAuth-2.0-based SSO. We confirmed that the attacks result in vPrograms not satisfying the ambient predicates.

Case #7 is about a relying party that uses the email address (email) field, rather than the claimed_id field, as the user's identifier. The reported vulnerability is because the signIn_req.email field can be excluded from the signature coverage by the malicious user, so that it bears an arbitrary value supplied by the client. We intentionally introduced this vulnerability to our OpenID 2.0 implementation on DotNetOpenAuth. When the attack is launched, the resulting vProgram fails to verify clause (3) in the ambient predicate, because final_result is taken from the signIn_req.email field, which is havocked in the vProgram.

Cases #3-#5 are about Interspire, which is another merchant software providing similar functionalities as NopCommerce. We have not applied CST on Interspire. However, based on the nature of the attacks, it is clear that they fall nicely into the scope of CST:

- Case #3 is an attack in which the attacker starts two independent transactions – one is expensive, the other is cheap. The attacker only performs the PayPal payment step in the cheap transaction, but not in the expensive transaction. At a particular stage, the merchant takes a signed orderID as the input argument. It is at this stage where the attacker supplies the signed orderID of the expensive transaction into the HTTP session of the cheap transaction, so the expensive order is checked out although only the cheap order is paid. This attack will be defeated by CST, because the orderID is always attached with the SymT, and signed together. When the attacker swaps the orderID of the expensive transaction into the cheap transaction, the SymT of the transaction has to be swapped in as well. The SymT clearly indicates that no payment step has been performed, so the ambient predicate will fail to verify.
- Case #4 is a vulnerability because the merchant may take the orderID from the client's cookie. According to our definition, any unsigned value supplied by the client must be havocked. A havocked orderID would prevent the ambient predicate from being verified.
- Case #5 is because the payment total is calculated based on the shopping cart at the checkout time, but the order being checked out is generated based on the shopping cart after the payment is made. The ambient predicate will not verify in this case, because the shopping cart is a runtime object, querying its property at two time points are semantically two method calls, corresponding to two different symbolic values. The equality would not be established in the verification.

Case #9 is about JanRain SSO service. The attack is to set the redirection destination (i.e., Redir_dest) to the attacker's website when the (victim) user tries to sign into a (victim) website. The JanRain server correctly checks the redirection destination, but the most important step in the attack is that the client can swap in an unchecked URL as the redirection destination after the checking. If CST was applied, the unchecked URL would be essentially a havocked value (i.e., the attached SymT would not indicate any logic constraint imposed on this URL), so the clause in the ambient predicate about the redirection destination would fail.

Similar to case #7, case #10 is about a relying party that intends to use the email field as user ID. However, the developer mistakenly uses an arbitrary non-email field as the email field, due to a misunderstanding of the OpenID 2.0 protocol. CST would prevent the flaw because the IdPAuth dictionary on the identity provider would not even contain this arbitrary field.

Cases #12-#14 include every exploitable flaw reported in [24]. In case #12, the attacker replaces the payee account ID with his own PayPal account ID, and checks out an order from the victim store by paying himself. Cases #13 and #14 are similar to case #3, in which the attacker places two orders in two sessions, and supplies a message obtained from the session of the cheaper order into the session of the more expensive order. We have explained that these are precisely the type of logic flaws that CST would prevent.

Cases that are not addressed by CST. CST relies on every party to correctly verify signatures. Case #6 is a vulnerability in signature verification. It is out of scope of CST. The root cause of case #8 is a client-side cross domain issue. Specifically, it is due to a special Adobe Flash communication mode that does not conform to the same-origin policy. This causes secret data from the IdP to be obtained by a malicious webpage on the victim user's browser. CST does not address security flaws in the underlying platform.

6.2 Protocol agnosticism

The fact that we check the same ambient predicates for systems adopting considerably different protocols shows the protocol agnosticism of their security goals. To make the point even stronger, we built implementations that *blatantly violate protocols but are nevertheless secure*. Three of them are shown in Figure 12.

Implementation (A) does not conform to OAuth 2.0 protocol (see Appendix C2-middle) for two reasons:

- *First*, the protocol requires token_req to contain the field AppSecret, which is a secret the identity provider assigned to every relying party at the registration time. We realized that the purpose of AppSecret is to prevent another website from impersonating the relying party to access the identity provider, which seemed unrelated to SSO. We removed AppSecret, and the ambient predicate still held, suggesting that AppSecret is indeed unnecessary in SSO. Note that we do not claim that AppSecret is useless in general in the OAuth protocol (in fact, we now understand precisely where it is useful).
- *Second*, the protocol requires the identity provider to check redirect_uri. In implementation (A), the identity

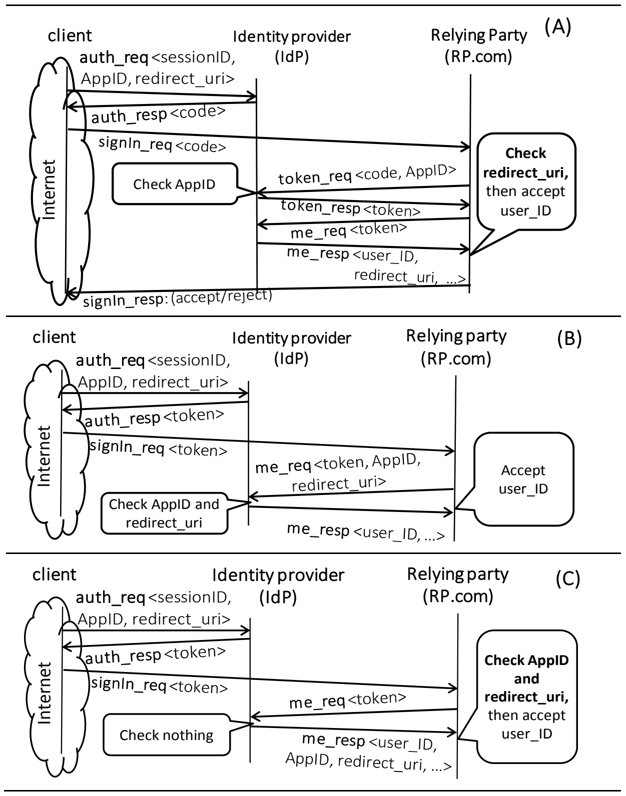


Figure 12: Protocol-violating yet secure implementations

provider does not perform the check. Instead, it returns redirect_uri in me_resp, so that the relying party can check it. The ambient predicate still hold in this case.

Implementation (B) even more blatantly violates the protocol, because it gets rid of code, but uses token for the client to authenticate into RP.com. According to a previous study, using token to authenticate is a pervasive and serious vulnerability [28]. Recently, the OAuth 2.0 specification has been augmented to explicitly forbid this kind of token usage (see section 10.16 of RFC 6749). However, we realized that this usage is vulnerable only because when the relying party accepts a token rather than a code, the steps token_req and token_resp will be skipped. The relying party directly calls me_req:(token), so the checking of redirect_uri and AppID required by the ambient predicate is missing. If me_req took additional arguments redirect_uri and AppID, and the identity provider performed the checking, as shown in implementation (B), then security would still be achieved.

Suppose the identity provider insists not to check anything, is the implementation doomed flawed? Not necessarily. The checking can be performed by the relying party, like in implementation (C).

Summary. We can see that all these implementations are just different ways of sharing the responsibility of performing all necessary checks. The OAuth 2.0 protocol describes one particular way, but not the only way. Of course, we understand that protocol conformance not only affects security, but also modularity, deployability, interoperability, etc. We do not suggest implementers disregard protocols, but only argue that security can (and should) be ensured independently, because understanding "who should do what, and why" about protocol specifications can be very subtle.

6.3 Performance

A significant strength of CST is its near-zero runtime overhead. Table 2 provides the measurement results, obtained from a server with a 2.10 GHz CPU and a 3.5 GB RAM, running Windows Server 2008. The numbers fall into two categories: per transaction cost and one-time cost. The time spent on synthesizing and verifying a vProgram belongs to the one-time cost, because the caching amortizes the cost over all transactions on all users. In fact, developers themselves are most likely the users who actually pay for the cost during testing.

Per-transaction cost. For a non-certifier party, the only runtime overhead is to produce the SymT. The source code hash is a pre-computed constant for a given version, so the only overhead is a string concatenation. Also note that *CST incurs no additional signing operation*, i.e., any unsigned message in the original implementation will remain unsigned in the CST-enhanced implementation. For the certifier, the only per-transaction overhead is the cache lookup for the SymT. Obviously, the runtime overheads for both a non-certifier party and the certifier should be extremely small. We nevertheless did the actual measurements to confirm that, for every system we implemented, the per-transaction runtime overhead is too small to report.

Table 2. Performance overhead – per transaction and one-time costs

	Per-transaction cost		**One-time cost**		
	Runtime overhead	Average traffic overhead	Program synthesis using a local de-hash server	Program synthesis using a remote de-hash server	compilation, byte-code translation and verification
Live Connect SDK	≅ 0ms	106 B/SymT	3ms	568ms	18758ms
OpenID 2.0 on DotNetOpenAuth	≅ 0ms	119 B/SymT	5ms	409ms	15380ms
Facebook SSO using ASP.NET MVC 4	≅ 0ms	120 B/SymT	5ms	408ms	12090ms
NopCommerce with Amazon Simple Pay	≅ 0ms	78 B/SymT	2ms	450ms	15444ms
NopCommerce with PayPal Standard	≅ 0ms	105 B/SymT	8ms	190ms	10990ms
Coin tossing gambling	≅ 0ms	205 B/SymT	3ms	945ms	32477ms

The SymT field incurs traffic overhead for protocol messages. We measured the average traffic overhead per SymT field (shown as Bytes/SymT). Our implementations use SHA-1 (160 bits), RSA (384 bits) and UTF-8 for hashing, encryption and encoding.

One-time cost. The synthesis cost is measured for two situations – when the de-hash table is stored locally or on another server. The first one mainly indicates the computational time of the synthesis algorithm, which is within 5 milliseconds in each of our case. The second situation may be more beneficial in practice because it offloads the de-hash table to another server. Although the synthesis time is longer, since it is a one-time cost, it should not be a performance concern in practice.

The last column in Table 2 corresponds to the real heavy-lifting step in CST. It consists of C# compilation into .NET byte code, byte-code translation into Boogie code and verification of Boogie code. The time reflects the significant logic complexity for verifying a transaction consisting of realistic methods. In contrast, today, this significant logic reasoning is never conducted, and correctness is taken on faith.

7 Related Work

There is a rich body of literature about verifying security protocols themselves, which we do not discuss here due to the space constraint. Research is also conducted to address issues in protocol implementations. Existing approaches can be categorized as either *top-down* or *bottom-up*. The top-down approaches focus on generating or verifying implementations based on formal specifications of protocols. For example, Bhargavan et al. [5] verified a number of reference implementations of the InfoCard protocol. In their work, the protocol and the security specifications are written in high-level languages F# and WSDL. Bhargavan and Corin et al. [6][9] developed a compiler that can synthesize a protocol implementation from a high-level F# specification of multiparty transactions. The bottom-up approaches try to extract protocols from actual systems. Aizatulin et al. [1] proposed to use symbolic execution to convert a protocol implementation in C into its high-level model in the applied pi calculus. Bai et al. developed a technique to extract SSO protocols from HTTP messages of network traces [3]. The uniqueness of CST is that it performs static verification at runtime, which converts the harder obligation of verifying a system into that of verifying intended transactions.

Proof carrying code (PCC) [20] is a technology for a code consumer (e.g., an OS kernel) to examine whether the code from an untrusted producer (e.g., a kernel extension from a third-party company) is accompanied by a logic proof of desired safety properties. CST and PCC target different problems. CST does not have the "proof carrying" aspect of PCC, but interestingly has a "code carrying" aspect that enables the verification.

Our work has connections with logic-based access control. Research on access control logic focus on expressiveness, decidability and theorem-proving efficiency of different logic frameworks. Lampson et al. defined a decidable logic based on the "*speaks for*" relation [16]. Appel and Felten found that many access control scenarios need higher-order logic, which is more expressive, but usually undecidable. They proposed proof-carrying authentication (PCA) [2], motivated by the idea of PCC, to shift the proof obligation to requestors. Code-carrying authorization (CCA) [17] is a follow-up of PCA. CCA allows requestors to provide fragments of the reference monitor's code (in form of the spi calculus), rather than proofs as in PCA. Our work is clearly distinguishable from research on access control logic: (1) The certifier in a CST system is not a reference monitor; rather the computation being certified by the certifier is akin to a reference monitor; (2) The notion of proof in a CST system is about correctness of program behaviors and discharged using standard program verifiers, while proofs in access-control systems are based on custom axioms and inference rules about trust and authority.

Connections can also be drawn between CST and secure multiparty computation [30] and verifiable computation [12] in applied cryptography. However, the goal are very different from CST. Secure multiparty computation is to enable parties to jointly compute a function over secret data held by individual parties. Verifiable computation enables a weaker device to securely outsource computations to untrusted servers.

8 Conclusions

We show that CST is a practical approach to achieve protocol-agnostic security for real-world online services, and that protocol-agnostic security enables system implementations to safeguard themselves against flaws in the protocol and developers' misinterpretations of the protocol or misassumptions about other parties.

CST represents a paradigm shift for developers. Programming is less about conforming to a protocol, but more about explicating the computations to establish an end-to-end global property. From the security standpoint, protocols become advisory rather than mandatory. What is truly mandatory is the security predicate.

Unfortunately, today, security predicates are vague and confusing: protocols use different terminologies for same concepts; security goals are often implicit or buried in the step-by-step instructions of individual protocols. A valuable effort we envision is that different protocol committees (and API-providing companies) agree on common terminologies and draft a "meta-specification" to formally define ambient predicates for each *class* of protocols, which will enable developers to check end-to-end properties with little arbitrariness. As we show in the paper, this goal is achievable in reality.

Acknowledgment

We thank Martin Abadi, Tom Ball, John Douceur, Ulfar Erlingsson, Cormac Herley, Jaeyeon Jung, Jay Lorch, Madan Musuvathi, Ed Nightingale, Prateek Saxena, Nik Swamy, Helen Wang and Lidong Zhou for insightful feedbacks on the paper, Akash Lal and Mike Barnett for help on Corral and BCT, Rich Draves, Yuri Gurevich, Jon Howell, Mike Jones, Yi-Min Wang for valuable discussions.

References

[1] M. Aizatulin, A. D. Gordon, and J. Jurjens. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. ACM CCS, pages 331–340, 2011.

[2] Andrew Appel and Edward Felten. Proof-Carrying Authentication. ACM CCS 1999.

[3] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. NDSS 2013.

[4] Michael Barnett and Shaz Qadeer. "BCT: A translator from MSIL to Boogie." Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2012.

[5] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Nikhil Swamy. “Verified implementations of the information card federated identity-management protocol,” ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2008.

[6] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, James J. Leifer. "Cryptographic Protocol Synthesis and Verification for Multiparty Sessions". IEEE Computer Security Foundations Symposium (CSF), 2009

[7] Boogie: An Intermediate Verification Language. http://research.microsoft.com/en-us/projects/boogie/

[8] Cloud Security Alliance. "The Notorious Nine – Cloud Computing Top Threats in 2013". https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf

[9] Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, Karthikeyan Bhargavan, James Leifer. "Secure Implementations for Typed Session Abstractions". IEEE Computer Security Foundations Symposium (CSF), 2007

[10] DotNetOpenAuth. http://dotnetopenauth.net

[11] Tom FitzMacken. Using OAuth Providers with MVC 4. http://www.asp.net/mvc/tutorials/security/using-oauth-providers-with-mvc

[12] Rosario Gennaro, Craig Gentry, and Bryan Parno. "Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers". CRYPTO 2010.

[13] Dick Hardt. "The OAuth 2.0 Authorization Framework (RFC 6749)". http://tools.ietf.org/html/rfc6749

[14] Akash Lal, Shaz Qadeer and Shuvendu Lahiri. "A Solver for Reachability Modulo Theories". Computer Aided Verification, 2012

[15] Leslie Lamport, “Password Authentication with Insecure Communication”, Communications of the ACM, Nov. 1981

[16] B. Lampson, M Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems, theory and practice. ACM Trans. Comp. Sys. 10, 4, Nov. 1992.

[17] Sergio Maffeis, Martin Abadi, Cedric Fournet, and Andrew D. Gordon. Code-Carrying Authorization. ESORICS 2008.

[18] Microsoft. ASP.NET Sample for Live Connect OAuth SSO. https://github.com/liveservices/LiveSDK/tree/master/Samples/Asp.net

[19] Microsoft Corporation. MSDN article - Contract Class. http://msdn.microsoft.com/en-s/library/system.diagnostics.contracts.contract(v=vs.110).aspx

[20] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. USENIX Symposium on Operating Systems Design and Implementation, 1996

[21] Nieuwenhuis, R.; Oliveras, A.; Tinelli, C. (2006), "Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)", Journal of the ACM 53 (6), pp. 937–977.

[22] NopCommerce. http://www.nopcommerce.com/

[23] OpenID Foundation. "OpenID Authentication 2.0 - Final". http://openid.net/specs/openid-authentication-2_0.html

[24] Giancarlo Pellegrino and Davide Balzarotti. “Toward Black-Box Detection of Logic Flaws in Web Applications.” Network and Distributed System Security (NDSS) Symposium, 2014

[25] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, E. Jay. "OpenID Connect Standard 1.0 - draft 21". http://openid.net/specs/openid-connect-standard-1_0.html

[26] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. *IEEE Symposium on Security and Privacy*, 2011.

[27] Rui Wang, Shuo Chen, XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. *IEEE Symposium on Security and Privacy*, 2012.

[28] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. *USENIX Security*, 2013.

[29] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen, InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations, in Network & Distributed System Security Symposium (NDSS), February 2013

[30] Andrew Chi-Chih Yao: Protocols for Secure Computations (Extended Abstract). FOCS 1982

[31] A collection of online services enhanced by CST. https://sites.google.com/site/protocolagnostic/

Appendix A: The vProgram for Section 3.2 (Abbreviated)

```
using …;
using System.Diagnostics.Contracts;

…   //omitting some class definitions
public class PlaceOrderResp_PayReq
{  …   //defining the fields for this message type
}
public class PayResp_CompleteOrderReq
{  …   //defining the fields for this message type
}
public class Cashier
{  public payment_record_t[] payments;
   public int i;
   …
   int getAvailableIndex()
   {   int i; havoc(i);
```

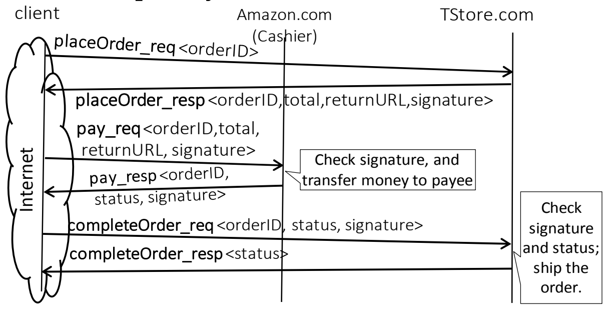
```
        Contract.Assume(0 <= i && i < payments.Length);
        return i;
    }
    PayResp_CompleteOrderReq pay(
                                PlaceOrderResp_PayReq req)
    {   i=getAvailableIndex();
        payments[i].payee = req.signer;
        payments[i].orderID = req.orderID;
        payments[i].total = req.total;
        PayResp_CompleteOrderReq resp;
        resp.redirectionURL = req.returnURL;
        resp.orderID = req.orderID;   resp.status = "Paid";
       resp.signer = "Cashier.com"; return resp;
    }
}
public class Merchant
{   public order_record_t[] orders;
    public string mySellerID = "John";
    public string myDomain = "TStore.com";
    …
    PlaceOrderResp_PayReq   placeOrder (PlaceOrderReq req)
    {   PlaceOrderResp_PayReq resp;
        int orderID = req.orderID;    resp.orderID = orderID;
        orders[orderID].status = "Pending";
        resp.redirectionURL = "https://Cashier.com/pay.aspx";
        resp.total = orders[orderID].gross;
        resp.returnURL =
                 "https://TStore.com/completeOrder.aspx";
        resp.signer = "TStore.com"; return resp;
    }
    bool completeOrder(PayResp_CompleteOrderReq req){
        if (req.signer != "Cashier.com" ||  req.status != "Paid" ||
              orders[req.orderID].status != "Pending")  return false;
        return true;
    }
}
class vProgram
{   ….
    static Merchant TStore=new Merchant();
    static Cashier MyCashier=new Cashier();
    static void Main()
    {
/* The vProgram for a normal transaction will contain L1 and
L2, but not L3. The vProgram for the attack described in
Section 3.1 will contain L3, but not L1 and L2.  */
L1:  placeOrder_resp=TStore.placeOrder(placeOrder_req);
L2:  pay_req = placeOrder_resp;
L3:  ~~havoc(pay_req);~~ //This line would fail the verification*/
      pay_resp = MyCashier.pay(pay_req);
      completeOrder_req = pay_resp;
      bool completeOrder_resp=
                 TStore.completeOrder(completeOrder_req);
      if (!completeOrder_resp) return;
L4:  Contract.Assert( placeOrder_req.orderID ==
                          completeOrder_req.orderID);
L5:  Contract.Exists(0, MyCashier.payments.Length, i =>
          MyCashier.payments[i].status == "Paid" &&
          MyCashier.payments[i].total ==
            TStore.orders[completeOrder_req.orderID].gross &&
          MyCashier.payments[i].payee ==
                            TStore.mySellerID     &&
          MyCashier.payments[i].orderID ==
                            completeOrder_req.orderID
      );
    }
}
```

Appendix B: Production rules for SymT

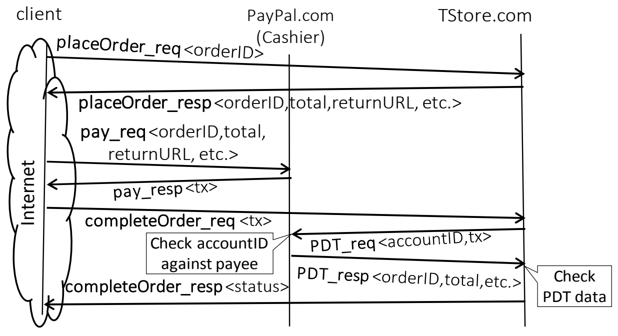
```
SymT            →  empty
                  | PARTY-ID : METHOD-CALL
                  | PARTY-ID :: METHOD-CALL
                  | PARTY-ID :: METHOD-CALL [ COVERAGE ]
ARGUMENT-LIST → SymT
                  | SymT , ARGUMENT-LIST
METHOD-CALL →  SRC-HASH (ARGUMENT-LIST)
                  | SRC-HASH ((ARGUMENT-LIST))
COVERAGE      → FIELD-NAME
                  | FIELD-NAME ,  COVERAGE
PARTY-ID       →  a.com | b.com | amazon.com | …
SRC-HASH    →  SHA-1 hash of source code
FIELD-NAME  →  name of a field in the output, e.g., x, y, z.
```

Appendix C1: Message diagrams for two checkout/payment flows in NopCommerce

Amazon Simple Pay

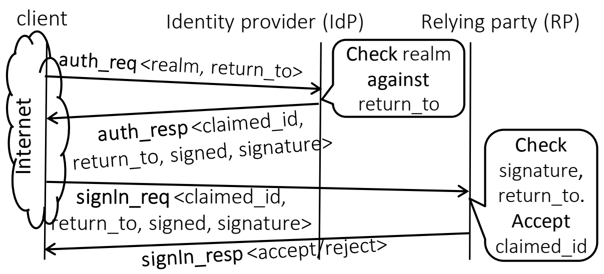

client
Amazon.com (Cashier)
TStore.com
Internet
placeOrder_req <orderID>
placeOrder_resp <orderID,total,returnURL,signature>
pay_req <orderID,total, returnURL, signature>
Check signature, and transfer money to payee
pay_resp <orderID, status, signature>
completeOrder_req <orderID, status, signature>
completeOrder_resp <status>
Check signature and status; ship the order.

PayPal Standard

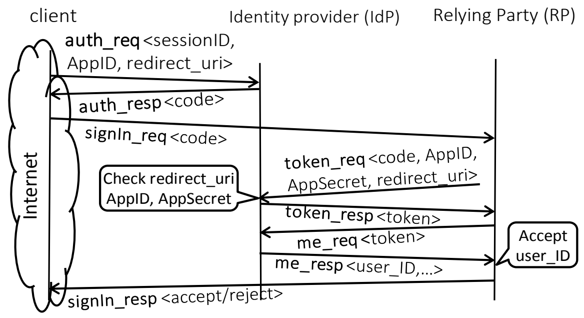

client
PayPal.com (Cashier)
TStore.com
Internet
placeOrder_req <orderID>
placeOrder_resp <orderID,total,returnURL, etc.>
pay_req <orderID,total, returnURL, etc.>
pay_resp <tx>
completeOrder_req <tx>
Check accountID against payee
PDT_req <accountID,tx>
PDT_resp <orderID,total,etc.>
Check PDT data
completeOrder_resp <status>

Appendix C2: Message diagrams for three SSO protocols

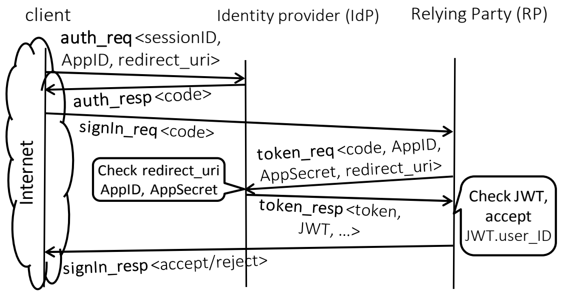
OpenID 2.0


client
Identity provider (IdP)
Relying party (RP)
Internet
auth_req<realm, return_to>
Check realm against return_to
auth_resp<claimed_id, return_to, signed, signature>
Check signature, return_to. Accept claimed_id
signIn_req<claimed_id, return_to, signed, signature>
signIn_resp<accept/reject>

OAuth 2.0



Live Connect


client
Identity provider (IdP)
Relying Party (RP)
Internet
auth_req<sessionID, AppID, redirect_uri>
auth_resp<code>
signIn_req<code>
token_req<code, AppID, AppSecret, redirect_uri>
Check redirect_uri AppID, AppSecret
token_resp<token, JWT, ...>
Check JWT, accept JWT.user_ID
signIn_resp<accept/reject>

Appendix D. Lines of code that we added or changed (comment and white lines excluded)

	The certifier	Shared methods (shared by the API-provider and the relying website)	The relying website	The API-provider
Live Connect SDK	347	0	48	100 (wrapper code of LiveID server)
OpenID 2.0 on DotNetOpenAuth		104	59	182
Facebook SSO using ASP.NET MVC 4		0	119	411 (wrapper code of Facebook server)
NopCommerce with Amazon Simple Pay		0	71	375 (wrapper code of Amazon server)
NopCommerce with PayPal Standard		0	71	239 (wrapper code of PayPal server)
Coin tossing gambling		283	45 for gambling; 83 for payment; 87 for token manager; 78 for coin tosser	375 (wrapper code of Amazon server)

Appendix E. Real-world cases studied in our security analysis

No.	Attack	Target system	Description of the attack/vulnerability reported in the literature	CST effectiveness
#1	Section III.A.1 of [26]	NopCommerce with PayPal Standard	The merchant does not check the total amount in the PDT response, so the shopper can pay an arbitrary amount.	Yes
#2	Section III.A.2 of [26]	NopCommerce with Amazon Simple Pay	The client makes a payment to MarkStore.com to check out an order from TStore.com (**explained in Section 3.1**)	Yes
#3	Section III.B.1 of [26]	Interspire with PayPal Express	The client places an expensive order and a cheap order, pays for the cheap one, and completes the expensive one.	Yes
#4	Section III.B.2 of [26]	Interspire with PayPal Standard	There is a situation where the merchant gets the orderID from the client's cookie. The client is able to change the cookie value to check out many orders with one payment.	Yes
#5	Section III.B.3 of [26]	Interspire with Google Checkout	The attack exploits an inconsistency between the total amount at the checkout time and the total amount when the order is to be completed.	Yes
#6	Section III.C of [26]	Websites using Amazon Payments	It is a signature validation bug.	No
#7	Section 4.1 of [27]	Websites using Google ID	The attack targets websites that use email address as the user ID. A malicious user can exclude the email field from the signature coverage of the identity provider to fool the relying party.	Yes
#8	Section 4.2 of [27]	Websites using Facebook Connect	It is a client side cross origin bug. The secret, i.e., Facebook token, can be obtained by a webpage from an arbitrary domain.	No
#9	Section 4.3 of [27]	Websites using JanRain sign-on	The attack is to cause the JanRain server to send the authentication secret (i.e., the identity provider's token) to an arbitrary website.	Yes
#10	Section 4.5 of [27]	Websites using Google ID	The flawed relying party mistakenly uses a non-email field as the email field to identify a user.	Yes
#11	Section 2 of [28]	Websites using OAuth implicit flow for SSO	The attacker targets websites that use OAuth access token to authenticate the user. **We explained it in Implementation (B) in Section 6.2**.	Yes
#12	Section IV.A.1 of [24]	osCommerce, CS-Cart and AbanteCart using PayPal Standard	The attacker replaces the payee account ID with his own PayPal account ID, and checks out an order from the victim store by paying himself.	Yes
#13	Section IV.A.2 of [24]	OpenCart and TomatoCart using PayPal Express	Similar to case #3, the attacker places two orders in two sessions. He records a message obtained in the session of the cheaper order, and supplies it in the session of the more expensive order.	Yes
#14	Section IV.A.3 of [24]	TomatoCart using PayPal Express	This vulnerability is essentially the same as case #13 above. The only difference is that, for TomatoCart, the attacker does not even need to pay for the cheaper order in To	Yes