2011 IEEE Symposium on Security and Privacy

How to Shop for Free Online
Security Analysis of Cashier-as-a-Service Based Web Stores

Rui Wang[1], Shuo Chen[2], XiaoFeng Wang[1], Shaz Qadeer[2]

[1] Indiana University Bloomington
Bloomington, IN, USA
[wang63, xw7]@indiana.edu

[2] Microsoft Research
Redmond, WA, USA
[shuochen, qadeer]@microsoft.com

***Abstract*— Web applications increasingly integrate third-party services. The integration introduces new security challenges due to the complexity for an application to coordinate its internal states with those of the component services and the web client across the Internet. In this paper, we study the security implications of this problem to merchant websites that accept payments through third-party cashiers (e.g., *PayPal*, *Amazon Payments* and *Google Checkout*), which we refer to as Cashier-as-a-Service or CaaS. We found that leading merchant applications (e.g., *NopCommerce* and *Interspire*), popular online stores (e.g., *Buy.com* and *JR.com*) and a prestigious CaaS provider (*Amazon Payments*) all contain serious logic flaws that can be exploited to cause inconsistencies between the states of the CaaS and the merchant. As a result, a malicious shopper can purchase an item at an arbitrarily low price, shop for free after paying for one item, or even avoid payment. We reported our findings to the affected parties. They either updated their vulnerable software or continued to work on the fixes with high priorities. We further studied the complexity in finding this type of logic flaws in typical CaaS-based checkout systems, and gained a preliminary understanding of the effort that needs to be made to improve the security assurance of such systems during their development and testing processes.**

Keywords- e-Commerce security; web API; Cashier-as-a-Service; logic bug; program verification

I. INTRODUCTION

Progress in web technologies has led to rapid growth of *hybrid web applications* that combine the *Application Programming Interfaces* (APIs) of multiple web services (e.g., search APIs, map APIs, payment APIs, etc.) into integrated services like personal financial data aggregations and online shopping websites. The pervasiveness of these applications, however, brings in new security concerns. The web programming paradigm is already under threat from malicious web clients that exploit logic flaws caused by improper distribution of the application functionality between the client and the server (e.g., relying on client logic to validate user privileges). The program logic of a hybrid web application is further complicated by the need to securely coordinate different web services that it integrates: failing to do so leaves the door wide open for attackers to violate security invariants by inducing inconsistencies among these services.

As an example, consider an online merchant integrated with the *Amazon Payments* service. The shopper's browser communicates with the merchant server to place an order, and with an Amazon server to make a payment. If the interactions between the two servers are not well thought out, the shopper may be able to shop for free. For instance, we discovered a real flaw where the merchant is convinced that the order has been paid for in full through Amazon while the payment has actually been made to the shopper's own Amazon seller account.

Intuitively, logic bugs related to multiple web services can be much more difficult to avoid than those in traditional single-service web applications – it is analogous to real-life experiences that when multiple parties discuss a subject by making individual one-on-one phone calls, it is generally difficult for each party to comprehend the whole picture. An honest party may say something out of context, or fail to understand another honest party's assumptions and reasoning, so a cheater is more likely to succeed in this situation than in a two-party conversation between the cheater and the only honest party. We will show many concrete findings to support this intuition.

Cashier-as-a-Service based checkout. As a first step towards understanding the security implications of multi-party web applications, we studied a category of online merchant applications that adopt third-party cashier services such as *PayPal*, *Amazon Payments* and *Google Checkout*. These cashier services, which we call *Cashier-as-a-Service* or simply *CaaS*, play a crucial role in today's e-commerce, since they act as a trusted third party that enables mutually distrustful parties to do business with each other. A CaaS can collect the payment of a purchase from the shopper and inform the merchant of the completion of the payment without revealing the shopper's sensitive data like a credit card number. A study showed that 59% of U.S. online shoppers would be more likely to buy in web stores that accept CaaS payment methods [8].

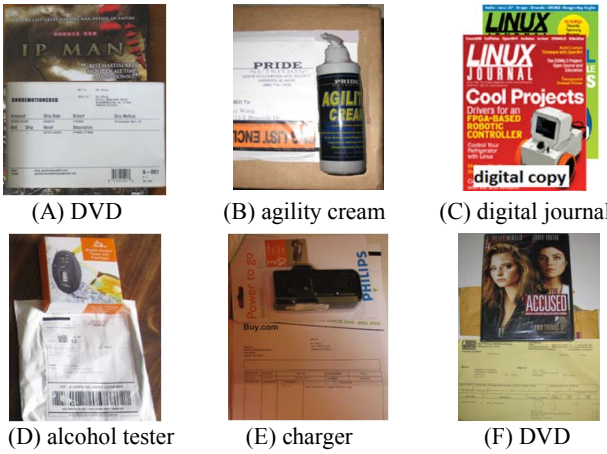
During a checkout process, communications happen between the CaaS and the merchant, as well as between these two services and the web client controlled by the shopper. This trilateral interaction is meant to coordinate the internal states of the merchant and the CaaS, since either party has only a partial view of the entire transaction. Unfortunately, the trilateral interaction can be significantly more complicated than typical bilateral interactions between a browser and a server, as in traditional web applications, which have already been found to be fraught with subtle logic bugs [9][12][16][36]. Therefore, we believe that in the presence of a malicious shopper who intends to exploit knowledge gaps between the merchant and the CaaS, it is difficult to ensure security of a CaaS-based checkout system.

Our work. The aforementioned concern turns out to be well-grounded in the real world. We conducted a systematic study of representative merchant software/websites that use the cashier services of PayPal, Amazon Payments and

1081-6011/11 $26.00 © 2011 IEEE
DOI 10.1109/SP.2011.26

Google Checkout. Our study revealed numerous security-related logic flaws in a variety of merchant systems, ranging from a high-quality open source software (*NopCommerce* [29]), to a leading commodity application (*Interspire* [20]), to high-profile merchant websites powered by closed-source proprietary software such as *Buy.com* and *JR.com*. Our attacker model is fairly simple – the attacker is a malicious shopper whose only capability is to call the web APIs exposed by the merchant and the CaaS websites in an arbitrary order with arbitrary argument values. We will show that everyone who has a computer and a small amount of cash (e.g., $25) is a qualified attacker. By exploiting the logic flaws, a malicious shopper is able to purchase at an arbitrarily-set price, shop for free after paying for one item, or even avoid payment.

To examine whether these logic flaws pose an imminent threat to e-commerce, we performed a responsibly designed exploit analysis on real web stores, including leading e-commerce websites such as Buy.com, and successfully checked out various items through exploiting these flaws. Figure 1 shows some of the items that were delivered to us, which included both physical and digital/downloadable commodities. This study was closely advised by a lawyer of our institution and conducted in a responsible manner, as elaborated in Section IV.



(A) DVD (B) agility cream (C) digital journals

(D) alcohol tester (E) charger (F) DVD

Figure 1: some received items and their shipping packages

While most of the logic flaws are due to lapses in the merchant software, we were surprised to find that well-known CaaS providers also need to shoulder responsibility: in particular, a serious error that we discovered in a set of Amazon Payments' SDKs has caused Amazon to significantly alter the way for verifying its payment notifications. We have reported our findings to all the affected parties, who acknowledged the significance of the findings and expressed gratitude for our help. We post part of our communications with them in [37].

To understand how complicated it is to ensure the absence of logic flaws in real-world CaaS-based checkout processes, we performed a formal verification study on a subset of Interspire's source code. We checked an invariant that is a conjunction of a series of bindings between order information and payment information. The outcomes turned out to be mixed: on one hand, formal methods did demonstrate their potential to address such a threat – they not only revealed all the flaws that we manually identified from the source code, but also new attacks that we did not expect. On the other hand, the complexity in the current checkout logic made even the state-of-the-art verifier hard to rule out the existence of potential logic flaws that can be exploited by more complicated attacks (with API-call sequences longer than what the current tool can explore). This suggests that little "margin of safety" can be offered by existing techniques for the exploits we discovered.

We view this work as a preliminary study that only touched relatively simple trilateral interactions, while other real-world applications may involve more parties (e.g., in marketplace and auction scenarios), and therefore can be more error-prone. This calls for further security studies about such complicated multi-party web applications.

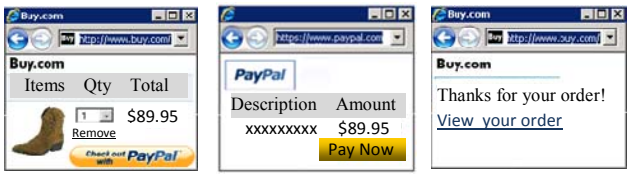
Contributions. Our contributions are summarized as follows:

- *In-depth security analysis of real-world CaaS-based checkout systems*. We performed the first systematic analysis of the security-related logic flaws in hybrid web applications. Our work discovers numerous security flaws in many representative checkout systems and demonstrates practical attacks that can happen to them. This suggests that there is inherent complexity in securely integrating multiple web services in a web application.
- *A preliminary analysis of the complexity of finding logic flaws in these systems*. We extracted the logic model from Interspire and analyzed it with a state-of-the-art verification-condition checker. From the study, we gained a preliminary but quantitative understanding of the inherent logic complexity of CaaS-based checkout systems.

II. BACKGROUND

A. Introduction to checkout workflows

Figure 2 shows some typical steps in a CaaS-based checkout. It starts when the button on page A of the merchant website (e.g., *Buy.com*) is clicked. In the figure, the button is "*Check out with PayPal*", so the click directs the shopper's browser to page B on PayPal (i.e., the CaaS), where he can click the "Pay Now" button to pay. Then, the shopper's browser is redirected back to the merchant's website to finish the order, which usually does not require the shopper's actions. Finally, the shopper gets the confirmation page C. The checkout process is arranged in this way to ensure that all three parties – the shopper, the CaaS, and the merchant, stay consistent despite their different locations across the Internet.



(A) click to place an order (B) click to pay in the CaaS (C) confirmation

Figure 2: some steps in a checkout workflow

What happens behind the scene here are HTTP interactions between the three parties, who communicate by calling web APIs exposed by the merchant and the CaaS. Such APIs are essentially dynamic web pages (denoted by diamond-shaped symbols in Figure 3), and are invoked through HTTP requests: the client sends an HTTP request through a URL with a list of arguments and receives an HTTP response (often a web page) dynamically constructed by the server as the outcome of the call. Throughout this paper, we refer to such a request/response pair as an *HTTP round-trip* or *RT*. In Figure 3, an RT is illustrated as a U-shaped curve, with its request arm labeled by the suffix ".a" and its response by ".b". The order in which different requests/responses happen is specified by both the numeric order of their corresponding RT labels and the dictionary order of their suffixes: for example, RT1.b comes before RT2.a but after RT1.a.a and RT1.a.b, and these last two messages are preceded by RT1.a, i.e., RT1.a → RT1.a.a → RT1.a.b→ RT1.b. Note that RT1.a.a is sent by the merchant during the handling of RT1.a, so RT1.a.a is not just chronologically after RT1.a, but causally depends on it. There is similar causality between RT2.a.a and RT2.a.

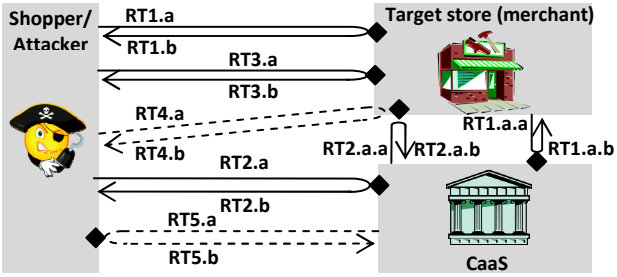


Figure 3: Web APIs and HTTP round-trips (RTs)

In the figure, RT1 and RT3 enable the shopper to invoke the APIs on the merchant and receive the responses. For example, RT1.a can be set off by a button click on page A in Figure 2, and RT3.b can carry the confirmation response (page C). RT2.a can be an API call to make a payment on the CaaS. It is sent when the "Pay Now" button on page B of Figure 2 is clicked. Moreover, RT1.a.a and RT2.a.a are sent by the merchant and the CaaS respectively to coordinate the state of the transaction with the other party. RT4 and RT5 will be explained later. These RTs serve as the building blocks for the workflows of various checkout solutions offered by different CaaS service providers (Amazon, PayPal, and Google). Some of the solutions, such as *PayPal Standard* and *Amazon Simple Pay*, are entirely based upon HTML, while the others, like *PayPal Express* and *Checkout By Amazon*, implement SOAP and NVP APIs.

We are not concerned with a network man-in-the-middle adversary intercepting RTs, because the checkout modules of all merchants and CaaS websites communicate exclusively over HTTPS to guarantee end-to-end security.

B. Challenges in securing checkout processes

To understand the nature of security threats that CaaS-based checkout systems are facing, the first step is to identify the security goal of these systems and the technical challenges in achieving it, which are described below.

Security invariant. The main security goal of a checkout system is to maintain the following *payment-completion invariant*: *Merchant M changes the status of an item I to "paid" with regard to a purchase being made by Shopper S if and only if (1) M owns I; (2) a payment is guaranteed to be transferred from an account of S to that of M in the CaaS; (3) the payment is for the purchase of I, and it is valid for only one piece of I; (4) the amount of this payment is equal to the price of I*. This invariant, though intuitive, implies a set of intertwined *binding relations* that should be respected in every step of the transaction. These bindings unequivocally link the merchant to a piece of the item being sold, the price of the item to the payment the merchant receives, and the payment for this specific purchase to the shopper.

Complexity in preserving the invariant. To achieve this security goal, a checkout system is expected to preserve the aforementioned invariant throughout a transaction. This turns out to be nontrivial, particularly in the presence of two web services. Specifically, the challenges in keeping both servers in consistent states include, but are not limited to, the following:

- *Confusion in coordination*. Given their incomplete views of a transaction, the merchant and the CaaS need to work together to preserve the invariant. This, however, is often hindered by the partial knowledge each party has about the other: the code of their systems is often off-limits to each other; the CaaS typically provides nothing but vague descriptions of its operations. As a result, misunderstanding often arises on the security assurance either party offers. For example, a merchant may assume that every notification of a payment completion from the CaaS must be about one of his transactions, but the CaaS may not have this guarantee and may expect a merchant to verify it by itself, as we show in Section III.A.2.
- *Diversity in the adversary's roles*. The merchant and the CaaS expose their APIs to the public, which enables the adversary to play more diverse roles than just the shopper, and thus to gain a deeper involvement in the checkout process than he could in a more traditional client-server interaction. The shopper can directly invoke a merchant's APIs such as RT4 in Figure 3, which mimics the behavior of the CaaS; the shopper can also mimic a merchant to register with the CaaS a callback API, which will later be called, as illustrated by RT5.
- *Parallel and concurrent services*. Both the merchant website and the CaaS need to serve many customers, and a shopper can concurrently invoke multiple purchase transactions. This further complicates the trilateral interactions, opening avenues for cross-transaction attacks.
- *Authentication and data integrity*. Compared with the two-party web applications, authentication in a CaaS-based checkout system involves three parties and is thus

more difficult in avoiding authentication and data integrity breaches. For example, we found that the integrity of each message field is not a big issue, but how to protect the bindings of the fields in different messages deserves careful thought processes and is the real pitfall.

In the next section, we show how real-world systems fail to answer to these challenges, indicating the urgent need to study the systematic solution to this problem.

III. Security Analysis of Real-world Merchant Applications and CaaS Services

In this section, we report our analysis of two popular merchant applications: *NopCommerce* [29] and *Interspire* [20], and their interactions with leading CaaS providers: PayPal, Amazon Payments and Google Checkout. Based on the insights from the study, we further probed for logic flaws in stores that run closed-source proprietary software.

Methodology. Our analysis follows an API-oriented methodology that dissects a checkout workflow by closely examining how individual parties can affect the arguments of the web API calls exchanged between them, and how these arguments affect the internal states of these parties. Some arguments of a web API carry the data flows between two parties, e.g. `gross`, `merchantID`, while others touch on their control flows, e.g., `returnURL`, `cancelURL`, and callback URLs that play a similar role to that of a return address or a function pointer in C/C++ programs. These arguments may not originate from the party that initiates the call. For example, the CaaS may use some data supplied by the shopper to communicate with the merchant through calling its APIs.

In our research, we studied whether the merchant/CaaS interactions in a checkout system present the malicious shopper opportunities to exert improper influence on the API arguments exchanged between these two services. To this end, we use a simple approach to keep track of the data that the adversary generates or can tamper with.

Table I lists the rules for labeling and tracing such data. Particularly, Rule (iii) makes the adversary the owner of any unsigned value that he sends, even though the value actually originates from other parties. All figures that we show in this section follow these labeling rules, which help describe the checkout workflows clearly.

TABLE I. LABELING RULES FOR API ARGUMENTS

(i) A newly generated value is labeled by its message origin – `T` for the target merchant under attack, `C` for the CaaS that the merchant uses, and `A` for the attacker/shopper;
(ii) A signed argument `arg` is labeled as $\texttt{arg}^{S*}$, where `S` is the signing party (`T`, `C` or `A`). Signed arguments are passed on across different parties without changing their origins;
(iii) Any unsigned value sent by the attacker is relabeled as `A`, regardless of the origin of the value.

To make succinct figures in the paper, we represent every URL in the following format:

*[https://]**host/apiName?arg1**[=value]**&…&argN**[=value]*

We often omit the "https://" prefix because all messages are HTTPS traffic. The string after "?" is the argument list. Usually we omit the concrete values of the arguments, but when a particular concrete value needs to be explicit, we provide the name/value pair as *argN=value*.

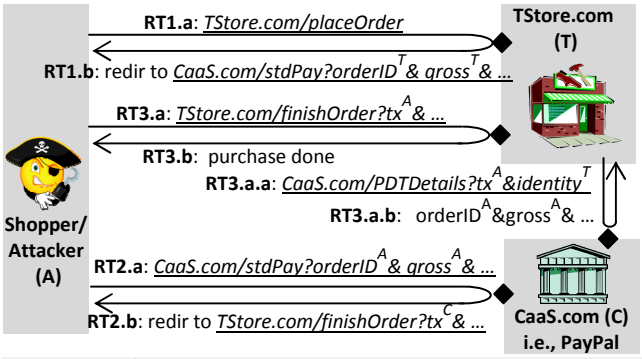
Limitation: CaaS as blackbox. Currently we do not have the source code for the CaaS side, but only the source code on the merchant side, including the merchant software and the CaaS' SDKs (Software Development Kits) compiled with the merchant software. For a CaaS service, we could only observe its concrete inbound and outbound messages, without knowing its internal logic, which might have subtle flaws as well. Therefore, what we have found only constitute a subset of the problem space.

A. Open-source software – NopCommerce

NopCommerce is the most popular .NET-based open source merchant software [29]. It was recently nominated as one of the best open-source e-commerce applications [34].

1) Integration of PayPal Standard – paying an arbitrary amount in PayPal to check out from the victim

PayPal Standard is the simplest method that a merchant website can integrate as its payment service. It is supported by NopCommerce. Figure 4 shows the workflow.




```
TStore.com/placeOrder: orderID=InsertPendingOrder ()
TStore.com/finishOrder:
  tnDetails=wCall_PDTDetails(tx,identity); /*resulting in
                                        RT3.a.a and RT3.a.b*/
  orderID=GetOrderIDField(tnDetails);
  order=LoadOrderByID(orderID);
  if (order≠null) and (order.status==PENDING)
        order.status = PAID;
```

Figure 4: NopCommerce's integration of PayPal Standard
(Note: RT3.a.a/RT3.a.b happen after RT3.a and before RT3.b)

First, the shopper clicks on the checkout button to send `RT1.a` to invoke the merchant's API `placeOrder`, which inserts the order information into a database, including the gross amount and the order ID. Since the order is unpaid, its status is set to PENDING. Then the merchant's response `RT1.b` passes the order information (e.g., `orderID` and `gross`) back to the shopper and redirects his browser to the CaaS (i.e., *CaaS.com/stdPay*), where the shopper pays according to the order information that his browser passes to the CaaS. The CaaS records the payment details and returns

`tx` as the transaction ID for the payment in `RT2.b`.[1] After the payment is done, the shopper's browser calls the merchant API `finishOrder` to finalize the invoice (`RT3.a`). Here we present the pseudo code of the function to highlight the part of its functionality of interest to us. More specifically, it makes a call to *CaaS.com/PDTDetails* (i.e., `RT3.a.a`), using `tx` and an authentication field `identity`, to get the payment details through `RT3.a.b`. Based on `OrderID` in the payment details, it looks up the order from its database. Once the order is located and its status is found to be PENDING, the status is set to PAID and a confirmation is sent to the shopper in `RT3.b`. In this entire workflow, no message field is signed (i.e., no "*" in any label in the figure). Security is expected through RT3.a.a and RT3.a.b, which are between the two servers.

Flaw and exploit. From Figure 4, a logic flaw is easy to see: the gross of the payment to CaaS is labeled as `A` using our analysis method, but the logic of `finishOrder` does not check the gross, which can be freely modified by the attacker. Therefore, setting the payment gross to an arbitrary value in `RT2.a` would not cause any trouble for the order to get through all the checkout steps.

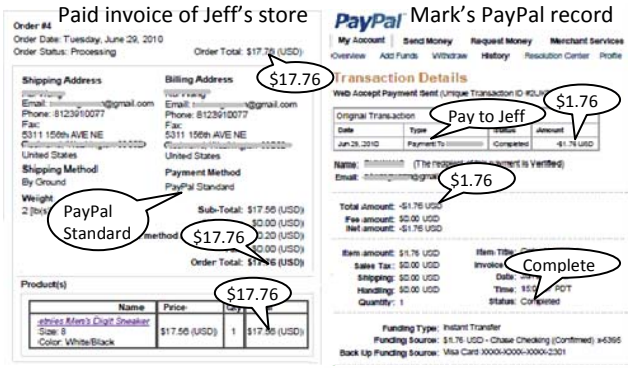
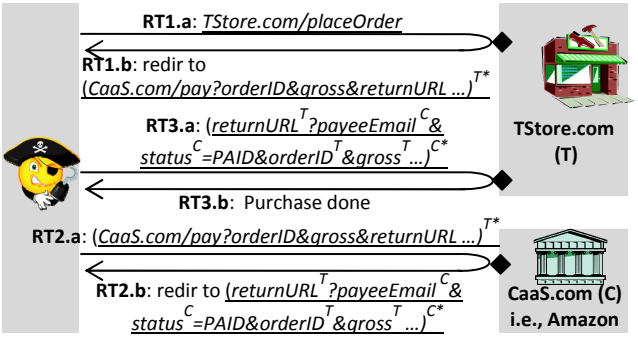


Figure 5: Jeff's paid invoice and Mark's PayPal record

To evaluate the practical feasibility of this attack, we installed NopCommerce on a server in our lab to set up a store for *Jeff*, and then registered a shopper *Mark* with PayPal. Figure 5 shows Jeff's finalized invoice and Mark's PayPal record. The price of the merchandise is $17.76. Exploiting the above flaw, Mark was able to pay $1.76 to complete the checkout. Interestingly, Jeff's invoice actually showed a payment of $17.76. There was no indication that the real payment was $1.76. In Section IV.A, we report our test of this exploit on a real store.

2) Integration of Amazon Simple Pay – paying to the attacker himself to check out from the victim

NopCommerce also supports Amazon Simple Pay, in which all messages after `RT1.a` are signed (`*`-labeled in Figure 6), so the shopper cannot tamper with the messages as in the prior example. Figure 6 shows the steps of this checkout method. `RT1.b` is used to redirect the shopper's browser to the payment API of the CaaS, passing `orderID`, `gross` and `returnURL` as the arguments. This message is signed by the merchant (labeled T*), so the shopper cannot tamper with the arguments when sending `RT2.a`. After the CaaS (i.e., Amazon) verifies the merchant's signature, the shopper makes the payment, which the CaaS records to its database (again, we omit a few RTs in the figure). The payee is the merchant who signs `RT2.a`, which, in Figure 6, is *TStore.com*. Then, by `RT2.b`, the CaaS redirects the shopper back to the merchant using `returnURL` that the merchant supplies in `RT1.b`. In NopCommerce, the URL is set to *TStore.com/finishOrder* for invoking the merchant API `finishOrder`. The entire message of `RT2.b` is signed by the CaaS, which is verified by the merchant. This checkout procedure seems secure: in Figure 6, no data can be contaminated by the attacker, i.e., nothing is `A`-labeled.



TStore.com/placeOrder: orderID=InsertPendingOrder ()

TStore.com/finishOrder **(handler of RT3.a)**:

```
if (verifySignature(RT3.a) ≠ CaaS) exit;
if (GetMsgField("status") ≠ PAID) exit;   /*payment status*/
order= GetOrderByID(orderID);
if (order==NULL or order.status ≠ PENDING) exit;
order.status=PAID;
```

Figure 6: NopCommerce's integration of Amazon Simple Pay

Flaw and exploit. Interestingly, this integration turns out to be vulnerable when the malicious shopper also plays the role of a different merchant. Specifically, anyone can open a seller account on Amazon, so can the attacker (in Section IV.B, we show that all the attacker needs is $25 cash for buying a MasterCard gift card from a supermarket; other personal information like name, email and phone number can all be faked). Suppose that the seller account is registered under the name "Mark". What the attacker wants to do is to pay Mark (actually, himself) but check out an order from a store belonging to Jeff (*https://jeff.com*).

The attack proceeds as follows. Acting as "Mark", the attacker drops `RT1.b`, but makes the message `RT2.a` by signing it using Mark's signature (labeled as `A*`):

(*CaaS.com/pay?orderID&gross&**returnURL**=https://jeff.com/finishOrder...*)^{A*}

The trick here is that the message signed by `A` actually carries a `returnURL` to Jeff (*jeff.com/finishOrder*). As a

[1] For the simplicity of presentation, we omit a few round-trips between RT2.a and RT2.b, which correspond to a few user clicks.

result, even though Mark (the attacker `A`) is the party that receives the payment, the CaaS will redirect the shopper's browser (`RT3.a`) to Jeff with a redirection to call `finishOrder`: redir to (*jeff.com/finishOrder?payeeEmail&status=PAID &ordered&gross…*)^{C*}

Although the message is indeed sent to Jeff, it is actually about the payment that the attacker made to Mark. The logic in `finishOrder`, as sketched in Figure 6, does not verify that the payment was made to Jeff, and therefore is convinced that the order has been paid.

Fundamentally, the problem comes from the confusion between the merchant and the CaaS about what has been done by the other party. An analogy can be drawn here to a real-life scenario in which Jeff first lets the shopper forward a signed letter to the CaaS: "*Dear CaaS, this shopper should pay $10 for order#123. When he pays, write a signed letter to Jeff. Thanks,* [Jeff's signature]" Later, Jeff indeed receives a response signed by the CaaS "*Dear Jeff, the $10 payment for order#123 has been received. I am talking about Mark's order#123 (nothing to do with you).* [CaaS' signature]." There are two important aspects to the misunderstanding that causes this security flaw. First, the CaaS thinks that it is fine to notify Jeff of Mark's transaction. Second, given the context of the conversation, Jeff believes that the response from CaaS is related to his original letter. Therefore, Jeff only checks that certain parts of the response (e.g., `orderID`, `gross`) match one of his pending orders. Because of this misunderstanding, even though all the messages between the two services are properly signed and verified, the binding between the order and the merchant is still broken.

Given the format of `RT3.a`, the only chance for Jeff to detect the attack is to check `payeeEmail`. Every merchant is required to provide an email address when opening an Amazon seller account. The address is included in `RT2.b` as part of the payment detail. Unfortunately, neither the CaaS nor the merchant application intend to use this email address for a security purpose: the CaaS never spells out the need to check this information, and the merchant software like NopCommerce and Interspire does not even ask for the email address at installation time.

B. *Commercial Software – Interspire*

Interspire shopping cart is one of the leading e-commerce applications, being used by more than 15,000 businesses across 65 countries [20]. Its hosting service, BigCommerce [6], was rated #1 e-commerce software for 2010 and 2011 by *TopTenReviews.com* [35]. The license fee of Interspire shopping cart software is $199. The source code package is available to its licensees.

1) *Integration of PayPal Express – paying for a cheap order to check out an expensive one*

Interspire incorporates over 50 payment methods of all major CaaS providers. Its integrations of these payment methods are typically more complex than those in NopCommerce. A prominent example is the way it uses PayPal Express [2], as illustrated in Figure 7. During a checkout, the merchant makes two calls to the CaaS. The first one is to inform the CaaS of an upcoming payment (`RT1.a.a`) with proper authentication data (`identity`). The CaaS then acknowledges the message with a `token` string for identifying this payment transaction, which the merchant passes to the shopper (`RT1.b`). The shopper then presents `token` to the CaaS, sets and confirms certain information about the payment (again, we represent these steps as a single step `RT2.a`). After that, the CaaS redirects the shopper's browser to the merchant API `finishOrder` with `token` and `payerID` as arguments (`RT2.b`, `RT3.a`). The code of `finishOrder` directly contacts the CaaS to complete the payment (`RT3.a.a`), and then lets the browser call the merchant API `updateOrderStatus`, which updates the status of the order (`RT3.b`, `RT4`). Note that some messages in this checkout process are not signed, which is not a security weakness, as the merchant directly verifies the data integrity with the CaaS (`RT3.a.a`).

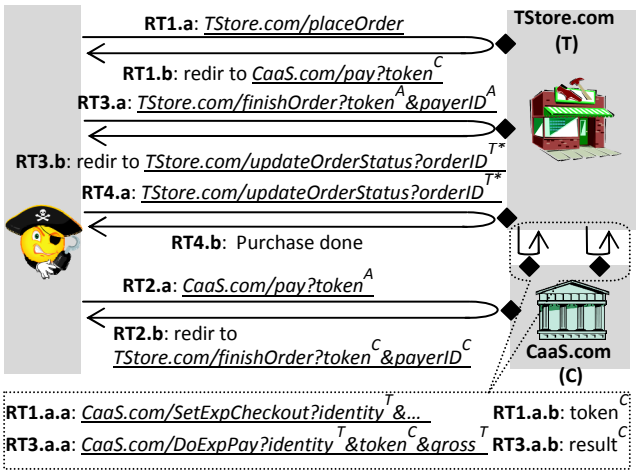


Figure 7: Interspire's integration of PayPal Express

Table II presents the pseudo code of `finishOrder` and `updateOrderStatus`. In `finishOrder`, the real payment is done by calling `wCall_DoExpPay`, which contacts the CaaS through `RT3.a.a` and `RT3.a.b`: if identity and other payment information is valid, the CaaS records the payment and returns `result = SUCCESS`. This result is saved in the session variable `SESSION["result"]`, a persistent variable that keeps the state of a shopper on the merchant website throughout his login session. At this point, the payment is complete, and the merchant is supposed to update the status of the order through API `updateOrderStatus`. Because the browser needs to be in sync with the merchant state, the merchant cannot directly call this merchant-side API, but needs to redirect the shopper's browser, passing `orderID` as an argument to the API `updateOrderStatus`. To prevent the shopper's tampering, `orderID` is first signed

[2] For the simplicity of description, we here focus on the most interesting part of the checkout procedure, ignoring some less important details.

by the merchant in `finishOrder`, and the signature is later verified within `updateOrderStatus`. The merchant then retrieves the order from the merchant database using `orderID`, and sets the status of the order to "PAID" if the session variable (`SESSION["result"]`) of the shopper is `SUCCESS`.

TABLE II. finishOrder() AND updateOrderStatus()

```
finishOrder() {
 result=wCall_DoExpPay(identity,token,gross);
     //This results in RT3.a.a and RT3.a.b
  SESSION["result"]=result;
  signedOID=sign(orderID);
  redirect("/updateOrderStatus?"+ signedOID);
     //This results in RT3.b and RT4.a
 }
 updateOrderStatus() {
   Verify the signature of orderID^T* in RT4.a
   If verification fails, then exit;
   order=LoadOrderByID(orderID);
   if (SESSION["result"]==SUCCESS)
       orderStatus=PAID;
   SESSION["result"]=null;      }
```

Flaw and exploit. A problem here is that as long as a properly signed order ID can somehow get into a session in the SUCCESS state, `updateOrderStatus` will mark the order corresponding to the order ID as PAID, no matter whether it has indeed been paid for. Therefore, once the shopper manages to acquire a signed `orderID` of an unpaid and more expensive order (denoted by `orderID2`), he can replace $\texttt{orderID}^{T*}$ in `RT4.a` with $\texttt{orderID2}^{T*}$ so as to use his current session state (which is PAID) to cheat `updateOrderStatus` into changing the status of the more expensive order into PAID. This enables the shopper to pay for a cheap item but check out an expensive one. Here we show how this can be achieved.

We used two separate browsers, e.g., Internet Explore and Firefox, to launch two separate login sessions. In the first session, we selected a cheap item and followed all the steps until `RT3.b` was complete, but we held `RT4.a`. At this moment, `SESSION["result"]` of this session had been set to `SUCCESS`, since the payment was made. Then, in the second session, we selected an expensive item, placed the order (`orderID2`), but skipped `RT2.a`. This caused the payment process (`RT3.a.a`) to fail, which was reflected by the state of the second session. However, `finishOrder` still redirected the shopper's browser (`RT3.b`) to invoke `updateOrderStatus`. This revealed $\texttt{orderID2}^{T*}$ to us, so we could copy-and-paste this signed $\texttt{orderID2}^{T*}$ into `RT4.a` of the first session, and sent it to finish the checkout of the expensive item.

2) Integration of PayPal Standard – stealing a payment notification and replaying it many times

Unlike NopCommerce's integration of PayPal Standard in Section III.A.1, in which the merchant calls the CaaS to get payment details, Interspire adopts Instant Payment Notification (IPN), an HTTP message that the CaaS uses to notify the merchant of payment status. In Figure 8, this message is shown as `RT2.a.a`, which is sent immediately after the shopper makes the payment through `RT2.a`. To use this notification method, the merchant (*jeff.com*) needs to specify an IPN handler URL. Interspire embeds the URL of the handler in `RT1.b`, the message that redirects the shopper's browser to the CaaS through `RT2.a`: for example, Jeff's store may set the handler at *https://jeff.com/handleIPN*. When the CaaS invokes this handler through `RT2.a.a`, it signs the argument list. The handler verifies the signature, the order data and the payment data in the IPN before updating the order status. The pseudo code of `handleIPN` is shown in Table III. `RT3` is not very important in our discussion here.

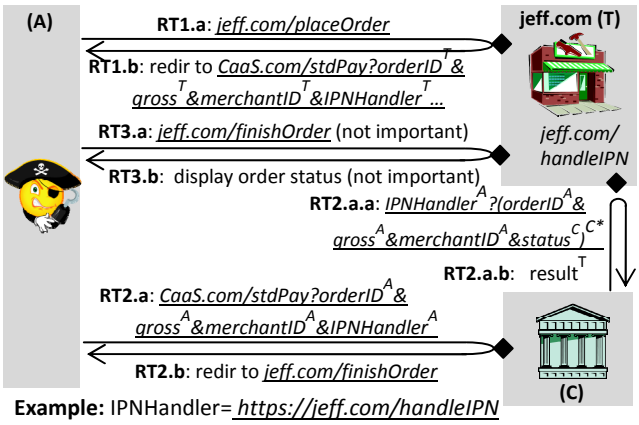


Figure 8: Interspire's integration of PayPal Standard

Flaw and exploit. `LoadOrderByID` is one of Interspire's heavily used utility functions. It is called in many situations, e.g., when handling a CaaS' request or handling a browser's request, therefore it is designed to be generic: when handling a CaaS request, e.g., in `handleIPN`, the function is called with an explicit `orderID`, as in line 1 of the code. However, a typical request from the browser, such as `RT3.a` above, does not contain the `orderID` field in the request URL. In this situation, `loadOrderByID(empty)` would be called, and the `orderID` is retrieved from a cookie named `ORDER_ID`.

TABLE III. PSEUDO CODE OF handleIPN()

```
handleIPN() {
1: order=LoadOrderByID(orderID);
2: if (order==null || order.status≠PENDING) exit;
3: if (merchantID ≠ Jeff's ID) exit;
4: if (gross≠order.gross || status≠PAID) exit;
5: order.status=PAID; }
loadOrderByID(orderId) {
  if (orderId is empty)
          orderId=COOKIE['ORDER_ID'];
  find order in database with orderId;
}
```

However, this generic design turns out to be problematic in PayPal Standard's IPN mechanism. The attacker can first change the message `RT2.a` by setting its `orderID` to be ***empty*** and setting `IPNHandler` to be

https://mark.com/handleIPN. This change causes PayPal's IPN message to be delivered to him via `RT2.a.a`, as illustrated in Figure 9.

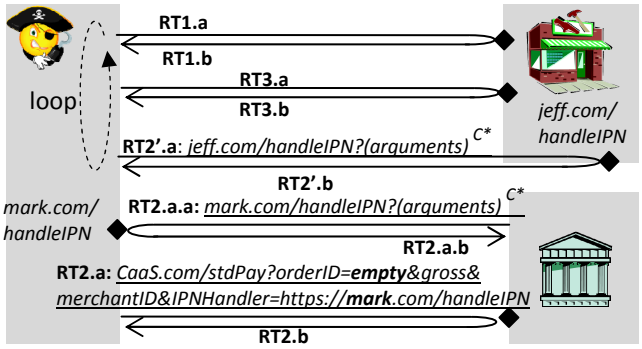


Figure 9: Multiple checkouts with one payment

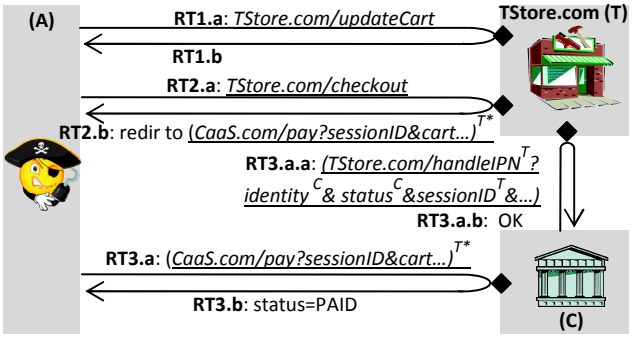
This move gives him an IPN message signed by the CaaS, which consists of the argument list `(orderID=`***empty***`&gross&merchantID&status)`^{C*}. Here we denote this string by `arguments`^{C*}. By replaying this message, the attacker is able to check out an arbitrary number of orders with the same prices: each time, all he needs to do is to place a new order by `RT1.a` (Figure 9), set the browser cookie `ORDER_ID` to be the ID of the order, then call Jeff's IPN handler with `arguments`^{C*} in `RT2'.a`, and then call Jeff's `finishOrder` by `RT3.a`.

In this exploit (Figure 9), the attacker plays all three roles: the shopper (`RT1.a` and `RT2.a`), the merchant (`RT2.a.a` for acquiring `arguments`^{C*}) and the CaaS (`RT2'.a` for replaying the signed IPN message). Of particular interest here is `RT2'.a` in which the attacker also changes his browser cookie, therefore it is a hybrid of a CaaS behavior and a browser behavior. This demonstrates how deeply the attacker can be involved in a CaaS-based checkout process and how complicated an exploit can be.

3) Integration of Google Checkout – adding items into the cart after the checkout button is clicked

Interspire's integration of Google Checkout contains about 4000 lines of code, the most complicated one among the four CaaS-integrations of the application we studied. Its simplified program logic is shown in Figure 10. Interspire utilizes several APIs to add/remove items in the shopping cart, which are aggregately denoted by `updateCart` (invoked by `RT1.a` in the figure) here for the simplicity of presentation. The checkout process (`RT2.a` to `RT3.b` in Figure 10) is triggered when the shopper clicks on the "Google Checkout" button. `RT3.a.a` is an IPN call made by the CaaS.

Flaw and exploit. A prominent feature of this checkout workflow is that no order is generated before the payment is made: the shopper is supposed to pay for the content of his shopping cart first; only when the merchant is informed by the CaaS via IPN (`RT3.a.a`) will the merchant's handler `handleIPN` create an order of the transaction according to what is inside the cart and set its status to "PAID", as illustrated in the figure. The problem here is that this procedure is not atomic: after receiving `RT2.b`, the shopper does not send `RT3.a` immediately. Instead, he can still call `updateCart` to change or add new items into his cart. Then, when `RT3.a` is sent, the current cart in the shopper's session is more expensive than the `cart` field in `RT3.a`. On the other hand, `handleIPN` loads the cart directly from the shopper's session, rather than from the CaaS, to build the order. This causes an inconsistency between what the CaaS sees in the cart at the pay time and what the merchant has at the checkout-completion time, so the shopper can pay for a cheap item, but check out many expensive items.



TStore.com/handleIPN:

```
1:  if (GetMsgField("status") ≠ PAID) exit;  /*payment status*/
2:  cart = LoadShoppingCart(GetMessageField("sessionID"));
3:  order = CreateOrder(cart);
4:  order.status=PAID;
```

Figure 10: Interspire's integration of Google Checkout

4) Integration of Amazon Simple Pay – avoiding payment

We discovered a bug that allows the attacker to fool the merchant into believing that a message sent by the attacker was generated by Amazon, and thus completely avoid payment. The details of the bug are described in [37].

C. Amazon Payments SDK flaw – interdependency of certificate authenticity and message authenticity

All the security flaws presented in the prior sections are directly related to merchant applications. The problem with CaaS providers is less clear, though they do need to better explain their operations and security assurance to avoid confusion on the merchant side. This, however, by no means suggests that the code of the CaaS is immune to this set of logic flaws: we did not perform an in-depth analysis on it just because the majority of it is not accessible to the public. From the small amount of the code the CaaS releases, we already discovered a serious flaw, as elaborated below.

Flaw and exploit. For all the messages bearing Amazon's signatures, the Software Development Kit (SDK) of Amazon Payments offers a signature verification API `validateSignatureV2`. This function, together with the rest of the SDK, is designed to be incorporated into merchant software. To verify signatures, the API needs to contact an Amazon certificate server to download Amazon's public key certificate. In our research, we found that a flaw

in the function enables the attacker to provide his own certificate to the merchant and thus to circumvent the verification. This vulnerability widely exists in various Amazon Payments SDKs, including *Amazon Flexible Payment Service*, *Amazon Simple Pay Standard*, *Amazon Simple Pay Subscriptions*, *Amazon Simple Pay Marketplace* and *Signature Version 1 to 2 Migration*. Most of them support five languages – C#, Java, PHP, Perl, and Ruby. It has been confirmed that they are all vulnerable.

Specifically, all URLs signed by Amazon Payments, such as an IPN message and the URL in a redirection response, have the following format:

(https://merchant/someAPI?arg1&arg2&...&argN&**certificateURL=https://fps.amazonaws.com/certs/090909/PKICert.pem**)^{C*}

The `certificateURL` field, which we omitted in the previous sections for simplicity of presentation, points to Amazon's certificate server for a certificate issued by VeriSign to Amazon. The entire URL is signed by Amazon (denoted as `C*`), including `certificateURL`. Thus, suppose the signature `C*` can be verified using the certificate referenced by `certificateURL`, it is reasonable in practice to say that if the message is signed by Amazon, then the certificate is an Amazon certificate, and vice versa. It seems to us that such an interdependency of certificate authenticity and message authenticity might have caused developers of `validateSignatureV2` to only verify the signature using the certificate referenced by `certificateURL`, without verifying the certificate itself.

To exploit this vulnerability, the attacker must act as a fake CaaS and use a server to store his own certificate. In our exploit, we used OpenSSL to generate a X.509 certificate, hosted it at *https://cert.foo.com*, which is a server under our control. Thus we can sign any URL as follows:

(https://merchant/someAPI?arg1&arg2&...&argN&
certificateURL =https://cert.foo.com/PKICert.pem)^{A*}

This signed URL, either used as a redirection URL or as an IPN, survives all checks in `validateSignatureV2`, and therefore allows the shopper to completely bypass Amazon Payments, to directly check out items from the merchant without pay. We have confirmed the feasibility of the attack on NopCommerce. In the next section, we report our communication with the development team of Amazon Payments on this flaw and their fix.

D. Popular stores running closed-source software

The source-code-based analysis on NopCommerce and Interspire, two of the most popular merchant applications, demonstrate that logic flaws in CaaS-based checkouts are indeed credible threats. Less clear here, however, is whether the unavaibility of merchant's source code can effectively conceal this type of logic flaws. To this end, we conducted black-box exploit analyses on two big stores, *Buy.com* and *JR.com*, based on general knowledge obtained earlier but without merchants' source code:

- *Buy.com flaw – shopping for free after paying for one item.* Buy.com is a leading online retailer with over 12 million customers in seven countries. It sells millions of products in various categories, including computers, cellular phones, software, books, movies, music, sporting goods, etc. It integrates PayPal Express as one of its checkout methods. Before the exploit analysis, we made a test purchase to capture the messages sent and received by the browser, and found that they are similar to those produced by Interspire's integration (Figure 7), though we could not observe the communication between PayPal and Buy.com, and the program logic on the merchant side.

Using our experience with Interspire's integration of PayPal Express (Section III.B.1), we evaluated the security protection of *Buy.com* through attempts such as changing the gross amount of an order, examining the way that signatures are used, etc. Despite initial failures, we discovered an effective exploit on *Buy.com*. As described in Section III.B.1, PayPal Express uses a *token* to uniquely identify a payment. We found that once the payment of one order is done, the shopper can substitute the token of this order for that of a different order (`RT3.a` in Figure 7). This allows the shopper to skip the payment step (`RT2.a`), but still convince Buy.com of the success of the payment for the second order.

Without access to the messages between Buy.com and PayPal (`RT3.a.a` and `RT3.a.b` in Figure 7) and the merchant-side code, we cannot conclusively determine what goes wrong with this checkout integration. Nevertheless, our study does confirm the pervasiveness of the logic flaws within checkout systems, which affect the coordination between integrated services, and the possibility of identifying and exploiting them even in the absence of the code of those systems.

- *JR.com flaw – attacker website selling items from JR.com at arbitrary prices.* JR.com is the online store of *J&R*, a well-known electronics retailer located in downtown New York City. The website accepts payments from Amazon's buyer accounts. Through studying the HTTP traffic of the browser and developer documentations provided by Amazon, we found that the payment method is *Checkout-By-Amazon* [31], which we did not investigate in our previous analyses of NopCommerce and Interspire.

A convenient way to integrate *Checkout-By-Amazon* is using the Seller Central form below, a toolkit provided by Amazon that automatically generates the HTML code for an Amazon-Checkout button for the item to sell.

To generate the HTML code, the seller first fills in information such as the item's name, price, and the seller SKU, etc. When the form is submitted, these fields, as well as a hidden field containing the seller's *merchantID*, are used by Amazon to produce the checkout button, whose

HTML code is signed by Amazon and can be cut-and-pasted onto the merchant web page selling the item.

Our analysis shows that again, the merchant and the CaaS fail to coordinate their security checks, which subjects this integration to the shopper's exploit. On one hand, Amazon does not fully prevent one merchant from creating a payment button for another merchant's item: the only information to tell the merchants apart is *merchantID*, which is public information and specified in a hidden field in the browser. On the other hand, like Interspire's integration of Google Checkout, *JR.com* does not create an order to bind an item to the price the shopper is willing to pay until the last step of the transaction, when the payment is complete, nor does it double-check the price at the payment-completion time. This allows the following attack:

Consider the attacker Mark who wants to buy an item *I* from *JR.com* at a price *X*. From the browser traffic corresponding to the Amazon-Checkout button for the item on *JR.com*, Mark can acquire the value of each field, including the hidden field *merchantID*. Then, he enters these values into the Seller Central form but changes the price to *X'*. To make the button point to *JR.com*, Mark also modifies the content of the hidden field, replacing his ID with that of *JR.com*. After that, he submits the form to acquire a signed checkout button from Amazon, which binds the price *X'* to JR.com's item *I*. Once Mark clicks on it, Amazon asks him to pay *X'* to JR.com, then uses a redirection to notify *JR.com* of the completion of the payment, which is accepted by the store.

IV. Exploit Analyses on Live Stores

In this section, we report our experiments on real-world web stores using CaaS services. The purpose of this study is twofold. First, we want to understand whether the vulnerabilities we discovered in merchant software can indeed be used against real online businesses, thereby posing a credible and imminent security threat; second, we hope to understand a number of aspects related to the exploits in real-world settings, such as how detectable the exploits are by regular auditing processes of the stores, how anonymous the attacks can be, and how various parties would respond to our bug reporting. To this end, we executed a series of exploit analyses within the ethical and legal boundary, as elaborated below.

Responsible experiment design. We carefully designed our evaluation strategy in order to carry out our experiments in a responsible manner. The entire study was conducted under the guidance of a lawyer at Indiana University. We strictly followed the principles below when performing exploits on real-world online stores: (1) we performed no intrusion of either merchant websites or CaaS services; (2) we ensured that no financial damage was inflicted upon the merchants involved, by canceling orders when possible, returning items, paying for unpaid balances, or placing orders in a special way (e.g., making two separate orders, one with a lower price and the other with a *higher* price); (3) we communicated our findings to the affected organizations and did what we could to help them improve their systems. Our responsible research effort was appreciated by these organizations.

A. Experiments on live online stores

Here we report our experiments conducted in various settings, ranging from open-source software on our server to closed-source systems on commercial websites, which demonstrates the credibility and pervasiveness of the threat.

Merchants on our server. We downloaded the latest version of NopCommerce (1.6), purchased the up-to-date licensed version (5.5.4) of Interspire, and installed these programs on our web servers. We also registered seller and shopper accounts with PayPal, Amazon Payments and Google Checkout. On the shopper side, we had Firefox and two HTTP debugging tools: Live HTTP Headers [21] and Fiddler [17]. Live HTTP Headers is a Firefox add-on capable of capturing and replaying HTTP/HTTPS traffic. Fiddler is a debugging proxy for intercepting and manipulating web traffic. Using these tools, we successfully executed all exploits described in Section III.

Our merchants on a commercial website. It came with little surprise that all exploits we discovered worked on the applications hosted on our server. However, when the same applications run on commercial websites, they could be configured differently and protected by additional security mechanisms. To evaluate the security threat in this more realistic scenario, we signed up a 15-day trial merchant account on *BigCommerce* [6], which is Interspire's hosting platform. Any user can register an account on *BigCommerce* to run his/her store powered by Interspire. Our evaluation showed that the same exploits also succeed against our store hosted on this platform.

Real merchants using Interspire and NopCommerce. All the security flaws reported in our analysis are related to the checkout and payment steps, which are only part of the entire purchase process. It is less clear whether end-to-end exploits in the real life would be caught by other fraud detection or account auditing procedures. In order to understand such end-to-end scenarios, we conducted exploit analyses on the following real online stores:.

- *GoodEmotionsDVD.com* is a NopCommerce-powered store that sells over 2,000,000 DVDs/CDs of movies, music, and games. It supports PayPal Standard. Exploiting the flaw in Section III.A.1, we were able to purchase a DVD at a lower price (Figure 1 (A)). We later paid the balance owed and notified the store and the developers about the exploit, and received their acknowledgement.
- *PrideNutrition.com* is an Interspire-powered store that sells nutrition supplement products. Its customers include athletic bodybuilders, licensed sports nutritionists, and certified personal trainers. The website provides PayPal Express based checkout. We bought a bottle of Agility

Cream for $5 less than the actual price, and received the shipment (Figure 1 (B)). We shared our discovery with the store, which expressed gratitude to our help [37].

- *LinuxJournalStore.com* is the online store of Linux Journal. It sells various Linux-related products, including T-Shirts, DVDs/CDs, magazines, and others. The store uses Interspire and enables PayPal Express, so it is vulnerable as we discovered. This time, we targeted digital products, which, different from physical commodity, do not need shipping. Today online commodities are often digital, e.g., electronic documents, memberships, phone-card minutes and game points. They are made available immediately after successful purchases. *LinuxJournalStore* sells digital Linux Journals in addition to paper ones. It accepts PayPal Express payments. We were able to pay for only one issue ($5.99) but check out two different issues ($11.98 together), and successfully download them (Figure 1(C)). In reference [37], we present our communication with the store.
- *LuxePurses.com*. Throughout our entire study, we placed at least 8 orders on real-world stores, including the orders described above and a few orders to be described later. Our purchase on *LuxePurses* was the only experience in which the store noticed the problematic payment. Our email communication is shown chronologically below:

> Email 1 from the store: *Mark, Thanks for your order. It will ship out later today and we'll send tracking info.*
>
> Email 2 from the store after several hours: *Hi Mark, Your payment via Paypal didn't complete for the full amount. The amount due, for this sale, was $27.15. You paid $17.41 through Paypal, which is $9.74 short. We will be invoicing you, for the $9.74 balance still owing through Paypal. Once it is paid in full, we will ship your item.*
>
> Email 3 from us: *I've paid the owed $9.74. Thanks.*
>
> Email 4 from the store: *Thanks so much! Our tech support team is confused as well! Seems to not have happened with anyone but us! We'll ship your item out tomorrow.*

Our order number was only "#175", which might suggest the low volume of the store's sales. Such a small order number and the above emails seem to indicate that they might have spotted the payment problem manually and accidentally, rather than due to a regular procedure.

Stores running closed-source proprietary software.

- *Buy.com*. We performed the exploit on Buy.com twice, and received an alcohol tester and a charger for free (Figure 1 (D)(E)). We contacted their customer service on our purchases. Although we were explicit about our exploit experiments, they could not understand the problems with our orders from their accounting data. Email 4 clearly indicates that their accounting system indeed believes that our order of alcohol tester, which is priced at $5.99, was paid, even though we did not pay at all. We returned the two items purchased after the refund period (45 days) expired to avoid being refunded, and continued to communicate our findings to the store.

> Email 1 from us: We explained that one of our orders, which costs $5.99, was unpaid, expressed the willingness of paying in full and provided them our credit card information.
>
> Email 2 from Buy.com: They misunderstood the situation, and sent us a generic reply explaining the possible reasons for delayed charging of credit cards, even though we paid through PayPal.
>
> Email 3 from us: *I am working on e-commerce security research. I bumped into an unexpected security issue about Buy.com's PayPal payments. I appreciate if you can forward this email to your engineering team. The finding is regarding the order 54348723. I placed the order in an unconventional manner (by reusing a previous PayPal token), which allowed me to check out the product without paying. I have received the product in the mail. Of course I will pay for it. Here is my credit card information [……]. Please charge my card for $5.99.*
>
> Email 4 from Buy.com: *Thank you for contacting us at Buy.com. Based on our records you were billed on 6/10/2010 for $5.99.*

- *JR.com*. We successfully placed several orders for different items with lower prices. They all reached the stage of pending fulfillment/shipping, before we canceled them (which was possible at this stage thanks to *JR.com*'s cancellation policy). We also placed an order for a DVD by setting a higher price and letting the shipping happen. The item was successfully delivered (Figure 1 (F)).

B. Attacker anonymity

Our research also shows that those attacks can happen without disclosing the attacker's identity. Here, we assume that the malicious shopper communicates through anonymity channels such as Tor or Anonymizer, which make his IP address untraceable.

Merchant/shopper anonymity. From three supermarkets in two U.S. states, we bought three $25 MasterCard gift cards by cash without showing any identity. We then visited the gift card website to register each card under "Mark Smith" at a random city. We confirmed that these cards were eligible for registering seller/buyer accounts on PayPal, Amazon, and Google, paying for orders, and receiving payments. To register these accounts, we also used fake identities to open a few Gmail accounts.

Anonymity in shipping. Purchase of digital items (e.g, memberships, software licenses, etc.) does not involve shipping, as the items become downloadable immediately after the payment is done. When it comes to physical items, the attacker needs to provide a valid postal address. However, the true identity of the recipient is usually not required: as an example, a USB driver we ordered was shipped to "Mark Smith" at our postal address through USPS. We guess that it may not be difficult for criminals to find addresses unlinked to them. When this happens, they can use fake identities to receive shipments.

C. Bug reporting and status of fixes

Besides communicating with the stores regarding the problematic purchases, we also shared technical details with affected stores, software vendors and CaaS service

providers, and offered assistance to improving their checkout systems. Here we present some of our efforts.

Amazon Payments. We reported the SDK vulnerability to the Amazon technical team, which immediately started an investigation. On 9/22/2010, 15 days after our reporting, new SDKs were released with an Amazon Security advisory acknowledging us [1]. In addition, Amazon announced that starting from 11/1/2010, 40 days after the advisory, Amazon servers would stop serving the requests made by vulnerable SDKs. All merchants must use the new version to verify signatures on Amazon's outbound messages, such as IPNs and redirections. Amazon is now working on the fix for the issue described in Section III.A.2 about Amazon Simple Pay.

LinuxJournalStore and Interspire. We disclosed to *LinuxJournalStore* the findings on its system. The store immediately contacted its software vendor — Interspire. Interspire developers were not able to figure out our attack based on their log data, so they approached us for details of the exploits. They recently notified us that these bugs were treated as top priorities, and have all been fixed in the latest version, and on *BigCommerce.com*.

NopCommerce. We reported the NopCommerce bugs to its developers. They have fixed the one related to PayPal Standard. The other bug (i.e. about Amazon Simple Pay), was left for Amazon to address, as we explained above.

Buy.com and JR.com. We have notified Buy.com four times and JR.com twice since October 2010, but have not received their progress updates.

V. Complexity Analysis of Checkout Logic

We have analyzed individual vulnerabilities and their real-world consequences. It is also important to study these instances as a class in order to understand the complexity of the overall problem in this space and obtain some quantitative measurements of the logic complexity.

A. The problem

We are interested in answering the following question: how complex is it for the developer of merchant software to detect *program logic flaws* that can be exploited by the malicious shopper to violate the payment completion invariant? We are particularly interested in exploits that induce inconsistencies between the transaction states perceived by the merchant and the CaaS. It is important to note that our focus is on program logic flaws, which are more design fallacies than coding flaws. This aspect distinguishes these flaws from vulnerabilities specific to programming languages (e.g., buffer overrun and cross-site scripting), operating systems and cryptographic primitives.

We consider an adversary whose only channels to interact with the merchant and the CaaS are the exposed web APIs. The adversary can invoke these APIs in an arbitrary order, set argument values for his calls at will, sign messages with his own signature, and memorize messages received from other parties to replay later, as long as the following rules are respected:

(1) The attacker is a registered customer of the merchant, and owns a payer account and a payee account on the CaaS;
(2) An API argument signed or under other integrity protection cannot be modified by other parties;
(3) The syntax of each API function must be followed.

The attacker being a web API caller implies that it does not have to behave like a normal browser, but can act as a merchant, a CaaS or any other entity that communicates through HTTPS. To understand the complexity of finding vulnerabilities exploitable by such an adversary, we conducted a formal reasoning study about Interspire's checkout logic, as reported in the rest of this section.

B. Modeling a subset of Interspire's logic

To investigate Interspire's logic for handling the four payment methods described in Section III.B, we first extracted a model from Interspire's source code corresponding to these handlers, then checked them against the payment-completion invariant using *Poirot* [30], an automatic verification tool that performs verification-condition (VC) generation and theorem proving.

Because the logic flaws that we focus on are language independent, our modeling language does not have to be a web programming language, such as HTML, JavaScript, ASP.NET or PHP, as long as it accurately preserves the program logic. Currently, our model is a program expressed in C language, which models the interactions between the merchant, the CaaS, and the malicious shopper: the three real-world parties are three modules in our program. The source code and full details for reproducing our results are given in [38]. Its components are illustrated in Figure 11.

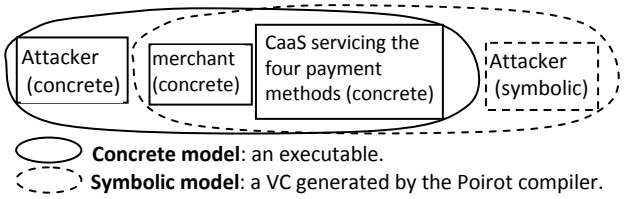


Figure 11: Concrete and symbolic models

Merchant and CaaS. The portion for modeling the merchant and the CaaS contains 506 lines of code. Table IV shows how certain key concepts of the actual application are modeled in our program.

TABLE IV. Real-world concepts mapped to our model

In actual systems	In our model
Merchant and CaaS servers	Merchant and CaaS modules
Web APIs	Functions annotated as `wAPIs`
URLs	Function or function pointers
HTTP round-trips (RTs)	Function calls
Signed message fields	Variables of type `SignedObject`

The merchant module in our program was directly transformed from the source code of Interspire, with the program elements in the original code replaced with the C code according to Table IV. In the absence of the source code on the CaaS side, we built its module based upon the specifications of its APIs, with a focus on the security-

related call arguments and other parameters as described in Section III.B. We also emulated the signing operation on API arguments using a special type `SignedObject`, which describes a signed data item with a pair of fields, `Obj` and `signer`. To indicate the item is signed, its content was copied into `Obj`, and the signing party was recorded in `signer`. This "signing" of course has no cryptographic strength, but since we only want to examine the program logic, this is sufficient for our definition of the payment-completion invariant, which is:

- *If the attacker is not allowed to create any* `SignedObject` *bearing the* `signer` *field* TargetStore *or* CaaS*, and can only call the functions annotated as* `wAPIs`*, is it always true that whenever an order is marked* PAID*, there is always a corresponding correct payment completed in CaaS?* (We will explain what constitutes "a corresponding correct payment" later.)

The attacker. In the C program, we implemented two attacker modules, one concrete and one symbolic. The concrete module was compiled together with the code for the merchant and the CaaS to generate a normal executable. It executed normal checkouts as well as all the attacks described in Section III.B. This was used to perform a sanity check on our model, including the functionalities of the merchant and the CaaS, and all the exploits we discovered.

The symbolic module was to analyze the complexity of finding logic flaws. It is sketched in Table V.

TABLE V. A SKETCH OF THE SYMBOLIC ATTACKER CODE

```
#include "MerchantAndCaaS.h"
typedef struct
    { SignedObject * msg;  int msgType; } Knowledge;
Knowledge[100]  Knowledgebase;
void main() {      while (1)  call_a_wAPI(nonDet());
}
void call_a_wAPI (int  wAPI_ID) {
  switch (wAPI_ID) {       //we have modeled 10 wAPIs
   case 1:    /*call placeOrder(), see RT1.a of Figure 8 */
       paymentType=nonDet();
       Merchant_placeOrder(paymentType);
       break;
   case 2:  /*call paypal's stdPay() , see RT2.a of Figure 8 */
      orderID= nonDet();  gross= nonDet ();  recipient= nonDet ();
      if  (nonDet ())  IPNHandler= TargetStore_PPLStdIPNHandler;
                  else  IPNHandler= Attacker_PPLStdIPNHandler;
       PPLStd_MakePayment(orderID,gross,recipient,IPNHandler);
       break; …
   case 10: …
   }
}
wAPI void Attacker_PPLStdIPNHandler(SignedObject * obj) {
   //handling RT2.a.a of Figure 9
   addToKnowledgebase(obj, PPLStdIPN);
} …
```

The idea is to let the attacker, i.e., the malicious shopper, repeatedly invoke the `wAPI` functions (emulated web APIs) on the merchant and the CaaS modules, using symbolic arguments, which was assigned the non-deterministic value "`nonDet()`". The symbolic attacker was compiled by Poirot to analyze for violations of the payment-completion invariant. As illustrated in Table V, the whole attacker module is organized as an infinite loop: each iteration uses `call_a_wAPI(nonDet())` to non-deterministically select a web API to call. Inside the implementation of `call_a_wAPI`, we also assign symbolic values to arguments of each wAPI. For example, consider the code under case 2 in Table V, which is used to call the API *https://paypal/stdPay* (See `RT2.a` in Figure 8). Some arguments of the call, including `orderID`, `gross` and `recipient`, are directly assigned symbolic values, while the value of `IPNHandler`, which can be either PayPal's handler or the attacker's, are chosen according to a symbolic value. Once all the arguments are set, the attacker calls `MakePayment` of PayPal Standard.

When the attacker module gets return values of wAPI calls (or its own wAPIs are called), it simply ignores the return values (or the argument values of incoming calls) if the values do not carry any signed data; otherwise (e.g., in the attacker function `Attacker_PPLStdIPNHandler`), it only needs is to record the signed data for later use. Note that in the current pseudo code, we define the return type `void`, which omits possibilities of exploiting bugs by sending error responses (e.g., `RT2.a.b` is not `OK`). In a more faithful model that aims at covering the error handling logic, the function should return a nondeterministic value.

C. Automatic verification

Poirot first compiles the symbolic model (consisting of the symbolic attacker along with the concrete merchant and CaaS) into an intermediate language, generates a verification condition (VC) based on the payment-completion invariant, then verifies the VC by a theorem prover. As mentioned earlier, the invariant requires that whenever an order is changed to the PAID state, there should be a "corresponding correct payment" record in the CaaS. This is interpreted in our current implementation as the situation when the gross of the payment matches the order's gross, its payee is the merchant, and its record is not matched by that of any previous order. Note that this invariant is only a necessary-yet-insufficient condition for a secure checkout: particularly, the invariant does not bind a product (an item) to the merchant who owns it, and as a result, exploits like the one that happens to *JR.com* could not be discovered. Nevertheless, our study reveals a lower-bound of the complexity for verifying the model.

By setting how many times Poirot should unroll the loop in function `main()`, we can control the depth of Poirot's search effort. We call this setting the *bound*. Bound x means that Poirot only considers all the execution paths that contain x or less web API calls.

Finding attacks. We ran Poirot on our model to automatically analyze all four payment methods that we studied manually. By setting the bound to 6, Poirot captured all the logic flaws discussed in Section III.B. The analyses took 355, 328, 381 and 330 seconds for PayPal Standard, PayPal Express, Amazon Simple Pay and Google Checkout.

It is particularly interesting that our analysis also

discovered new and more efficient attack avenues. For example, we thought that the attack on Interspire's PayPal Express (Section III.B.1) must be launched through two sessions (e.g., through IE and Firefox as described in the section); the attack instance reported by Poirot, however, only needed one session. We performed this new attack on the real Interspire executable, which was found to work exactly as indicated by the tool. The details of this exploit is given in [37], due to the space constraint of this conference version. What is important here is that it demonstrates that the formal reasoning approach seems promising in getting insights about the program logic that we focus on.

Empirical analysis of the complexity. We hypothetically fixed the logic flaws in the model, so that we can measure the complexity of each bounded verification, i.e., to verify no attack possibility within each bound. Table VI gives two complexity metrics: the number of *conflicts* the theorem prover encountered and the total time for verification, in the shaded rows and the clear rows, respectively. When a theorem is being proved, there are many Boolean decisions to explore. For each decision point, the theorem prover takes one branch and goes deeper into the search. A conflict happens when the theorem prover needs to backtrack and take the second branch of the decision point. Conflicts are the most important reason for the state explosion in the search; therefore, the number of conflicts is a good indicator of the complexity of verification[3]. The time measures were based on our PC specification: Intel Core 2 Duo CPU 3.00 GHz, 4GB memory, 80GB hard disk.

Table VI shows that both metrics increase significantly with the bound. For bound 7, most verifications encountered out-of-memory errors (OOM). The last row is for the verification of the APIs for all four payment methods. This best reflects the complexity in the actual implementation of Interspire, which currently has no mechanism to prevent the attacker from calling all APIs that belong to all payment methods. In this scenario, the verification for bound 6 already ended with an OOM.

TABLE VI. NUMBER OF CONFLICTS AND TIME FOR EACH BOUND

	1	2	3	4	5	6	7
PayPal Standard	167	574	1.3K	4.4K	42K	574K	OOM
Total time in seconds	15.2	48	103	253	385	3645	OOM
PayPal Express	33	247	595	1.3K	4.1K	29K	229K
Total time in seconds	16.1	42	85	145	225	379	1492
Google Checkout	120	479	1K	3.2K	26K	324K	OOM
Total time in seconds	14.9	44	92	156	302	2295	OOM
Amazon Simple Pay	123	523	1.3K	6K	74K	1636K	OOM
Total time in seconds	14.5	49	113	193	476	15113	OOM
All APIs	567	1.7K	4.5K	74K	2313K	OOM	OOM
Total time in seconds	21.5	156	258	926	17384	OOM	OOM

D. Implications of the complexity analysis results

Our measurement data seem to indicate a few interesting points for developers:

1) Automatic verification is necessary. On one hand, tools exist today to find flaws in extracted logic models, as we empirically demonstrated. On the other hand, manual verification of its security is really hard. Hundreds of thousands of backtracks in the reasoning process are involved, well beyond what human brains can handle.
2) Application developers should help lower the complexity so that higher confidence can be achieved by bounded verifications. Currently, bound 6 is often the limit of our machine's computational power for individual payment methods, and bound 5 is the limit for all payment methods together. However, many of our known attacks already take 5 or 6 steps to accomplish, so the "margin of safety" is too small. We believe that some efforts can be taken by developers to lower the logic complexity, and thus to increase the margin of safety. For example, the payment methods should be strictly separated at runtime so that static verification only deals with each payment method individually. Also, annotating the code with pre- and post-conditions would make verifications much easier.

VI. PAYMENT PROTOCOLS VS. PAYMENT APIS

Secure payment protocols have been studied for a long time. Early efforts can be traced back to the dawn of the Internet age. Examples of these protocols include *iKP* of IBM and *STT* of Microsoft/Visa [18], as well as a number of digital cash protocols. Among them, the most well known is perhaps *Secure Electronic Transaction* (SET) [39] proposed by Visa and MasterCard, in collaboration with GTE, IBM, Microsoft, Netscape, RSA and VeriSign. The security properties of this protocol were partially checked through formal verification by many researchers, including Bolignano [10], Lu et al [23], Meadows et al [25] and others. Formal analyses [19] were also performed on other payment protocols, such as NetBill [13] and DigiCash [11].

However, to the best of our knowledge, none of these protocols was deployed on the Internet and used by real-world e-commerce systems. The technologies that are actually adopted by today's e-commerce are web services like PayPal, Amazon Payments and Google Checkout, which are never referred to as "payment protocols". Indeed they are not protocols – they are APIs with proprietary implementations and public interfaces, accompanied by the developer's guides and sample code. Compared with protocols, which clearly specify the actions different parties are supposed to take, the ways these APIs are used are less rigorously defined, thus offering flexibility to their callers. Presumably, the flexibility contributed to the programmer friendliness and thus the popularity of these payment APIs. However, it leaves the security of today's checkout systems contingent upon the merchant-side program logic, which is less disciplined. How to securely call APIs has always been a challenge in programming, not specific to web APIs. For example, *strcpy* and *setuid* in C are notoriously difficult to call securely. In this sense, it is not a surprise that CaaS APIs leave plenty of rooms for logic bugs in web stores.

[3] Poirot's runtime is proportional to the number of conflicts and the work done per conflict in theory reasoning. The explosive growth in the number of conflicts leads us to believe that the cost of theory reasoning is dwarfed by the cost of the backtracking search.

Perhaps our work suggests that it is worthwhile to revisit the possibility of payment protocols, assuming that lessons have been learned from the unsuccessful adoptions of the techniques. Of course, the effectiveness of a protocol adoption should be put in perspective. After all, security of a theoretically-proven protocol often depends on many factors in real systems. First, its incorrect implementation could bring in security bugs. Also the assumptions underlying its design can be totally different from actual operational settings. As an example, the designer of a protocol could ignore the facts that anybody (essentially with no real identity) is eligible to be a seller, or a real-world system actually needs to operate in concurrent HTTP sessions (Section III.B.1). Finally, security of the whole system is also contingent on how the payment module interacts with other modules, e.g., bugs could exist if the state of a shopping cart can be changed during the payment processing (Section III.B.3), or the order ID is retrieved from a client cookie (Section III.B.2).

We believe an important contribution of our work is that it provokes a soul searching in both academia and the e-commerce industry on the prior effort on building a secure and usable payment system, which should preserve APIs' flexibility, and achieve formally verified security guarantee.

VII. Related Work

Technologies on security protocol verification. For decades, techniques for verifying protocols' security properties have been the focus of many studies. Classic approaches can be grouped into two categories, according to Millen [26]. The first category is based on an algebraic model defined by Dolev and Yao [14]. Prominent examples of these techniques include Interrogator [26] and NRL Protocol Analyzer [24][15], in which protocol flaws are identified through searching a protocol's state space for the paths that lead to insecure states. They were successfully applied to detect previously unknown bugs in security protocols. The second category is based on an axiomatic system about protocol participants' *beliefs*, as formalized by Burrows, Abadi and Needham (a.k.a. BAN logic) [7]. The BAN logic is believed to be more limited than the Dolev-Yao model, but it is decidable. The approach was applied on a number of protocols, such as Kerberos, Needham-Schroeder public-key protocol, CCITT X.509, etc. It is worth noting that despite its proof, the Needham-Schroeder public-key protocol was later found vulnerable by Lowe under the man-in-the-middle assumption [22]. The field of protocol verification has been advanced significantly over years. Abadi's recent tutorial [2] covers many techniques. Some may not fit very well into Millen's classification, such as the approaches based on type systems [4].

Research has also been conducted on analyzing other security protocols, e.g., fairness and verifiability [33] of a contract signing protocol [3]. TulaFale is a specification language to describe SOAP-based protocols and thus to enable formal checking of security properties for web services [5].

Security issues in e-commerce. Security weaknesses and flaws in e-commerce technologies were discussed in various sources. Price manipulation bugs existed in some early shopping cart implementations, as reported in [32] and [27], which used the cart total stored in a browser cookie to generate the order. We found that today's leading shopping carts, e.g., every cart that we studied, could not be similarly attacked. Another shopping cart bug was reported in [9]. It allowed items to have negative quantities.

Also worth mentioning is a new payment protocol *3D-Secure*, which is promoted by Visa and also adopted by MasterCard. It is marketed under the names *Verified By Visa* and *MasterCard SecureCode*. A main goal is to protect a credit card with a password to foil card-not-present attacks (e.g., using a stolen card number). Murdoch and Anderson discussed a set of weaknesses in *3D Secure* [28], e.g., GUI design, registration procedure, privacy protections.

Technologies addressing web application logic bugs. Researchers have shown increased attention to logic bugs in web applications. The proposed technologies fall in two categories: (1) those helping avoid logic bugs in new applications (a.k.a., the secure-by-construction approach); (2) those finding logic bugs in legacy applications.

Examples of the technologies in category 1 include Swift [12] and Ripley [36]. They are both built upon distributing compilers, such as Google Web Toolkit and Microsoft Volta, which automatically partition a single web program between the server and the client. Swift views the security task as a "logic placement" problem. To tackle it, Swift allows the developers to annotate the source code for security requirements so that it can perform information flow analysis to decide what logic can be securely placed on the client side. Ripley views the task as a logic replication problem: it runs a server-side replica of the client-side logic so that tampering with the client would result in inconsistencies between the client and the replica. Technologies in category 2 for legacy applications include NoTamper [9] and Waler [16]. NoTamper detects parameter validation bugs by finding conditions checked only by the client logic but not the server logic. Waler is a technology to generate *likely-invariants* based on runtime traces, and checks the likely-invariants against the source code.

The aforementioned technologies addresses logic bugs in web applications architected as client-server or client-frontend-backend. Our work explicitly focuses on websites integrating third party web APIs. The logic bugs appear to be more elusive in this new context.

VIII. Conclusions and Future Work

We presented our analysis for Caas-based web stores, as an example of security challenges in third-party service integration. We found serious logic flaws in leading merchant applications, popular online stores and a CaaS provider (i.e., Amazon Payments), which can be exploited to cause inconsistencies between the states of the CaaS and the merchant. As a result, a malicious shopper can purchase

an item at a lower price, shop for free after paying for one item and even avoid payment. We reported our findings to the affected parties and received their acknowledgements. Our further analysis revealed the logic complexity in CaaS-based checkout mechanisms, and the effort required to verify their security property when developing and testing these systems.

We believe that our study takes the first step in the new security problem space that hybrid web applications bring to us. Even for the security analyses of merchant applications, we have just scratched the surface, leaving many intriguing functionalities (e.g., cancel, return, subscription, auction, and marketplace) unstudied. An interesting question might be, for example, whether we can check out a $1 order and a $10 order, and cancel the $1 order to get $10 refunded. We are also considering the security challenges that come with web service integrations in other scenarios, e.g., social networks and web authentication services. Fundamentally, we believe that the emergence of this new web programming paradigm demands new research efforts on ensuring the security quality of the systems it produces.

ACKNOWLEDGMENT

We thank Martín Abadi, Brian Beckman, Josh Benaloh, Cormac Herley, Dan Simon and Yi-Min Wang for valuable discussions, Akash Lal for important advices on Poirot, Beth Cate for the legal assistance and Robert Schnabel for the support that makes this work possible. We also greatly appreciate Trent Jaeger for shepherding. Authors with IU were supported in part by the NSF Grant CNS-0716292 and CNS-1017782. Rui Wang was also supported in part by a Microsoft Research internship.

REFERENCES

[1] Amazon Security Advisories. Amazon Payments Signature Version 2 Validation. https://payments.amazon.com/sdui/sdui/security

[2] Martín Abadi. Security Protocols: Principles and Calculi (Tutorial Notes), Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures, Springer-Verlag (2007), 1-23.

[3] N. Asokan, Victor Shoup, and Michael Waidner. Asynchronous protocols for optimistic fair exchange. In Proceedings of IEEE Symposium on Research in Security and Privacy, pages 86–99, 1998.

[4] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon. Modular verification of security protocol code by typing. ACM Symposium on Principles of Programming Languages (POPL), 2010

[5] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, Riccardo Pucella. TulaFale: A security tool for web services. In Symposium on Formal Methods for Components and Objects (FMCO), 2003

[6] BigCommerce. http://www.bigcommerce.com/

[7] Michael Burrows, Martín Abadi, and Roger Needham. 1990. A logic of authentication. ACM Trans. Computer Systems 8, 1, 18-36.

[8] Ecommerce Statistics Compendium 2010. http://econsultancy.com/us/reports/e-commerce-statistics/downloads/2076-econsultancy-ecommerce-statistics-uk-sample-pdf

[9] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan, "NoTamper: Automatically Detecting Parameter Tampering Vulnerabilities in Web Applications," ACM Conf. on Computer and Communications Security, 2010

[10] Dominique Bolignano. “Towards the Formal Verification of Electronic Commerce Protocols,” Proceedings of the IEEE Computer Security Foundations Workshop, 1997.

[11] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Proceedings on Advances in cryptology (CRYPTO '88).

[12] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zhen, "Secure Web Applications via Automatic Partitioning," ACM Symposium on Operating Systems Principles (SOSP), October 2007.

[13] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. 1995. NetBill security and transaction protocol. In Proceedings of the 1st conference on USENIX Workshop on Electronic Commerce (WOEC'95).

[14] Danny Dolev and Andrew C. Yao. 1981. On the Security of Public Key Protocols. Technical Report. Stanford University, Stanford, USA.

[15] Santiago Escobar, Catherine Meadows, and Jose Meseguer. 2005. A rewriting-based inference system for the NRL protocol analyzer: grammar generation, the 2005 ACM workshop on Formal methods in security engineering (FMSE '05). ACM, New York, NY, USA, 1-12.

[16] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," USENIX Security Symposium, August 2010.

[17] Fiddler Web Debugger. http://www.fiddler2.com/fiddler2

[18] Phillip M. Hallam-Baker. Electronic Payment Schemes. http://www.w3.org/ECommerce/roadmap.html

[19] Nevin Heintze, J. D. Tygar, Jeannette Wing, and H. Chi Wong. Model checking electronic commerce protocols. The 2nd USENIX Workshop on Electronic Commerce , Berkeley, CA, USA. 1996.

[20] Interspire Shopping Cart. http://www.interspire.com/shoppingcart

[21] Live HTTP Headers. http://livehttpheaders.mozdev.org

[22] Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. Information Processing Letters 56(3), 1995

[23] Shiyong Lu and Scott A. Smolka. 1999. Model Checking the Secure Electronic Transaction (SET) Protocol. The 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '99).

[24] Catherine Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. Journal of Computer Security, 1992.

[25] Catherine Meadows and Paul F. Syverson. "A Formal Specification of Requirements for Payment Transactions in the SET Protocol," Financial Cryptography 1998

[26] Jonathan K. Millen. The Interrogator Model. IEEE Symposium on Security and Privacy 1995..

[27] K. K. Mookhey, "Common Security Vulnerabilities in e-commerce Systems," http://www.symantec.com/connect/ articles/common-security-vulnerabilities-e-commerce-systems

[28] Steven Murdoch and Ross Anderson, "Verified by Visa and MasterCard SecureCode: or, How Not to Design Authentication," Financial Cryptography and Data Security, January 2010

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

[30] Poirot: The concurrency sleuth. http://research.microsoft .com /en-us/projects/poirot/

[31] Resources – Amazon Payments. https://payments.amazon.com/sdui /sdui/business/resources#cba

[32] SecurityFocus.com. "3D3.Com ShopFactory Shopping Cart Cookie Price Manipulation Vulnerability," http://www. Securityfocus.com/bid/6296/discuss

[33] Vitaly Shmatikov and John C. Mitchell, Analysis of a fair exchange protocol, Symposium on Network and Distributed Systems Security (NDSS '00), San Diego, CA, Internet Society, 2000.

[34] Softpedia, "Choose the Best Open Source CMS for 2010," http://news.softpedia.com/news/Choose-the-Best-Open-Source-CMS-for-2010-158440.shtml

[35] TopTenReviews. eCommerce Software Review 2011. http://ecommerce-software-review.toptenreviews.com

[36] K. Vikram, Abhishek Prateek, and Benjamin Livshits, "Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution," ACM Conference on Computer and Communications Security (CCS), Nov. 2009.

[37] Rui Wang, Shuo Chen, XiaoFeng Wang, Shaz Qadeer. “How to Shop for Free Online -- Security Analysis of Cashier-as-a-Service Based Web Stores”. Technical Report, IU-CS-TR690. Supporting materials are available at http://research.microsoft.com/~shuochen/caas/supp/

[38] Rui Wang, Shuo Chen, XiaoFeng Wang, Shaz Qadeer. “A Case Study of CaaS Based Merchant Logic,” http://research.microsoft.com/en-us/people/shuochen/caaslogiccasestudy.aspx

[39] Wikipedia, "Secure Electronic Transaction," http://en. wikipedia.org/wiki/Secure_Electronic_Transaction