Bachelor Thesis

Cashier-as-a-Service based Webshops Overview and Steps towards Security Testing

Daniel Hirschberger

Date: 26.09.2016
Supervisors: Prof. Dr. Jörg Schwenk
Dipl.-Ing. Vladislav Mladenov
M. Sc. Christian Mainka

Ruhr-University Bochum, Germany

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Homepage: `www.nds.rub.de`

Erklärung

Acknowledgements

First I would like to thank my parents for enabling me to study at the RUB which is quite a distance away from my hometown and for their continuous support. Next I would like to thank my study group and friends for keeping studying and leisure time fun. And finally I would like to thank my supervisors Jörg Schwenk, Vladislav Mladenov and Christian Mainka for their helpful input.

Contents

List of Figures

List of Tables

1. Introduction and Motivation

Shortly after the invention of email, if you wanted to have an email account, you had to setup your own mailserver. Admins soon offered to setup mail addresses for their users which they could use. The user then had to install a mail client and have it configured to work with the server. After the user understood how the mail client works, he could use his email account without further help.

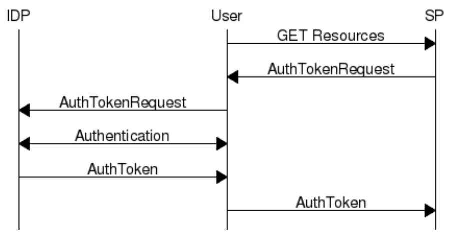
Nowadays, if somebody wants to have an email account, he can search the internet for mail providers and register an account on their own. It is no longer required to undergo the lengthy process of having an admin setup everything. This is because someone already setup a mail server and chose to provide users with mail accounts. Setting up a piece of software and providing others access to it is known as *Software-as-a-Service (SaaS)*.

Lately there is a paradigm shift in the nature of applications. You no longer have to install an email client to have access to your mail. Many email providers allow you to read, edit and send mails from an interface on their webpage. This is an outstanding example that shows that whole applications are being shifted to the server-side. Another example is Google Docs, which provides an office suite that is accessible from the webbrowser. Rich-clients are no longer needed because they are superseded by remote services which provide a user interface in the webbrowser.

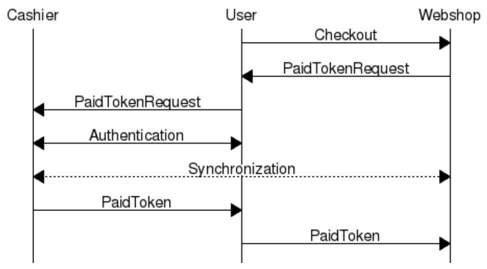
SaaS is not only used by users but by other applications as well. Developers no longer have to worry about handling user authentication because of *Single-Sign-On (SSO)* services or payment because of *Cashier-as-a-Service (CaaS)*. They can use these services provided by another party and concentrate on the development of their application.

Nevertheless caution has to be taken in developing an application which uses this services. Even if the service itself was secure, there is still the possibility that the newly developed application handles the responses from that service in a wrong way and thus enables attacks which can have devastating consequences, for example logging in with an account you do not own or having another user pay for your shopping.

One possibility of logging in via SSO is the *Service Provider (SP)* -initiated login. An overview of this process in given in Figure 1.1a. A user wants to use a service from the SP and is redirected with an AuthenticationTokenRequest to the *Identity Provider (IdP)*. If the user can successfully authenticate at the IdP, he is redirected with the AuthenticationToken to the SP. The SP recognizes the AuthenticationToken and can be sure that the user is authenticated. He consumes the token and logs the user into his service. The typical protocol flow for webshops that use CaaS is demonstrated in Figure 1.1b. A user is browsing through a webshop and adds items to his cart. Then he clicks



(a) SSO: SP-initiated Login



(b) CaaS: High-level Overview

Figure 1.1.: Overview of SSO and CaaS

the checkout button and is redirected with an PaidTokenRequest to the CaaS, where he has to authenticate. If this succeeds and he authorizes the transaction, he is redirected with the PaidToken to the webshop. The webshop sees the PaidToken and can be sure that the user paid. With this knowledge the webshop can finalize the order and begin the shipping process. Throughout this process the cashier and the webshop have to keep their application states synchronized. If this would not be happening, an attacker could trick the shop in several ways. In summary the cashier fulfills the role of the IdP, the webshop the role of the SP and additional steps have to be taken to keep the applications synchronized. Apart from that, the basic idea of the protocols are quite identical. Despite being similar there is a huge gap between SSO and CaaS concerning security research.

For SSO there are many protocols like *OpenID / OpenID Connect* [10] and *SAML* [21].
For every cashier that should be supported, the API of that specific cashier has to be implemented. The APIs differ in large parts from cashier to cashier. If the developer is lucky, there exists an SDK he can use to make this process easier. If this is not the case, he has to implement every single one himself. The fact that there is no common API or standardized protocol also means that security research is much harder.

The latest paper concerning the security of a SSO solution is from 2016 [9]. The latest paper about CaaS security is from 2011 [23].

It is quite ironic that CaaS is widely used yet barely researched. More so, considering that attacks on CaaS integrating webshops directly yield money or items.

This thesis is meant to provide a starting ground for the security analysis of CaaS based webshops and spark further research.

2. Overview

2.1. Overview of Webshops

We begin our collection of webshop software with search queries like `open source webshops`, `best webshop software`, `ecommerce software` and `comparison of webshop software` in several search engines. Apart from several names of such products, we also found popularity comparison sites like *BuiltWith* [2], *Datanyze* [6] and *Wappalyzer* [24]. We extracted the names of webshop software solutions by browsing through their popularity lists. Luckily we also found a shopping cart migration service called *Cart2Cart* [5], which provided us with even more names. After collecting a huge amount of names into a spreadsheet, we develop the structure of the spreadsheet with the columns which can be seen in Table 2.1.

We can acquire all of this information by visiting the homepage of the specific webshop and the usage statistic sites presented earlier.
We calculate the mean and maximum popularity across the statistics sites to gain a definitive criteria we can use when choosing the webshops we wish to analyze. The spreadsheet is by no means exhaustive and focuses on open-source solutions.

A shortened version of the spreadsheet showing only open source solutions ordered by the maximum popularity can be found in A.1.

2.2. Overview of Cashiers

We obviously need cashiers to set up an CaaS-integrating webshop. We will focus on one of the widely used cashiers, namely *PayPal*, *Amazon Payments* or *Google Wallet* because no prior research on their protocols has been done and critical flaws in them affect a huge audience. Nevertheless there exist many small local payment providers in various countries which could face similar problems. An example of such a payment provider is *paydirekt* [11] in Germany. For the three big cashiers we collect information on transaction fees and the possibility of sandbox accounts which ease our future analysis.

Product Information	**Popularity**	**Third Party Support and Comments**
Name	on *BuiltWith*	Link to supported cashiers/-payment methods
Link to Webpage	on *Datanyze*	Comments
Link to Download	on *Wappalyzer*	
Link to Documentation	Arithmethic Mean	
Open Source	Maximum	
Link to Source Code		
License		
Price		
Free Trial		
Programming Languages		

Table 2.1.: Columns sorted by type

2.3. Selection of Webshops and Cashiers

The only hard criteria we have for choosing a webshop is that it is open source and we can host our own instance of the software. We chose these criteria because we do not want to have unnecessary costs by using a commercial product and want to set it up on a virtual machine for analysis. From those which fulfill these conditions, we pick the two with the highest `Maximum Popularity` from our spreadsheet. This means we will set up *Woocommerce* with a maximum popularity of 31% and *Magento* which has a maximum popularity of 22%. We notice that the arithmetic mean of the popularity is not suited for comparison because not every webshop is present on all statistics site which has a huge effect on the mean.

The only real requirement for the cashier is that we can access developer or sandbox accounts that let us use the real API in an sandboxed environment. This means that we can do as many transactions as we like without losing real money and without facing any real consequences.
The cashier we integrate into the webshops is *PayPal* because it is widely used as evidenced by a study conducted in 2013 [25] and fulfills our requirements.

3. Methodology

Before we are able to analyze the payment protocols we have to recover the unknown protocol. The goal is to recover the essential messages of the protocol while skipping unnecessary requests which for example load javascript or fetch images. For this purpose we developed the following methodology.

We begin the analysis by checking out a product and completing the whole transaction. Then we enable the `intercept` option of the *Burp Proxy* and begin the checkout process anew. After each iteration we have to clear the cookies from our browser, otherwise we would obtain a different message flow which is not suited for analysis. At first we do not have a representation of the protocol. We drop the first request we have not yet included in our representation of the protocol. If the checkout does not work afterwards, we gain the information that this request is essential to the protocol and add it to our representation of the protocol. Note that a successful checkout is defined by the automatic completion of the order by the webshop. We repeat this for every request the client submits during the checkout process while ignoring requests for resources like javascript, images and the like.
The result of this step is the minimal amount of messages that has to be exchanged for a successful checkout.

The next step is to test which parameters are needed for the checkout process. Note that we do not analyze the authentication of the user at the cashier. We repeat the checkout process many times while stripping one parameter at a time from the request. If the checkout can be successfully completed, we learn that the parameter is not vital to the checkout and can be left out. As a result we know which parameters are essential for the checkout process.
In the next step we modify one of the vital parameters at a time and see if the checkout completes successfully. The result is the list of parameters which are essential to the checkout but can be manipulated.
The final result is the minimal representation of the protocol with all required parameters.

For the analysis of the protocols we let the *Web attacker* introduced in [1] slip into different roles and also use the *User Behaviour* presented in the paper:

1) Malicious user The attacker plays the role of a normal user who wishes to buy goods at a webshop.

2) Malicious shop The attacker plays the role of a seemingly benign shop that sells goods.

3) CSRF - Attacker In addition to the attacker-in-the-browser model we also allow a user to get tricked into issuing a GET request via phishing.

4. *Magento's PayPal's Express Checkout* Integration

4.1. Testing Environment

For testing we have a virtual machine running Ubuntu 14.04.4 LTS with Kernel 3.13.0-9 and PHP 5.5.9-1ubuntu4.19 in the NDS Cloud at our disposal. We have set up *Magento* in version 2.0.7 and added products with the following names and prices: *AES 1€*, *PKI 2€*, *SSL 10€*.
Furthermore we integrated *PayPal Sandbox Accounts* which we acquired by registering a private *PayPal* account and creating them on `developer.paypal.com`.

In order to analyze the protocol we have to be able to intercept all messages that are exchanged between the user, the cashier and the webshop. We chose the *Burp Suite* [4] as our primary analysis tool because of its unique capabilities and ease of use. Intercepting the traffic between the user and the webshop, and the user and the cashier is achieved by routing it through the *Burp Proxy* via setting the proxy rules in our browser to `localhost:8080`. Intercepting the traffic between the webshop and the cashier and routing it through our local *Burp* instance is not as easily accomplished.

The basic idea is to proxy the traffic between the webshop and the cashier through a local port on the virtual machine and then forward this port via SSHs remote port forwarding feature to our machine on which *Burp* is running. We create one proxy listener for each port we forward to distinguish the requests by port.
Luckily *Magento* provides an option in the admin interface to force API requests which are directed to *PayPal* through a proxy. Additionally we have to set the option `Enable SSL verification` to `No` so that *Magento* does not check the validity of the certificate and we can man-in-the-middle the connection to *PayPal*.

We create a proxy listener in *Burp* which is listening on port `8081` and connect the proxy port on the virtual machine with our local machine via SSH remote port forwarding by issuing the following command:

```
ssh -N -R 8081:localhost:8081 <IP of Virtual Machine>
```

Figure 4.1 illustrates the finished setup.

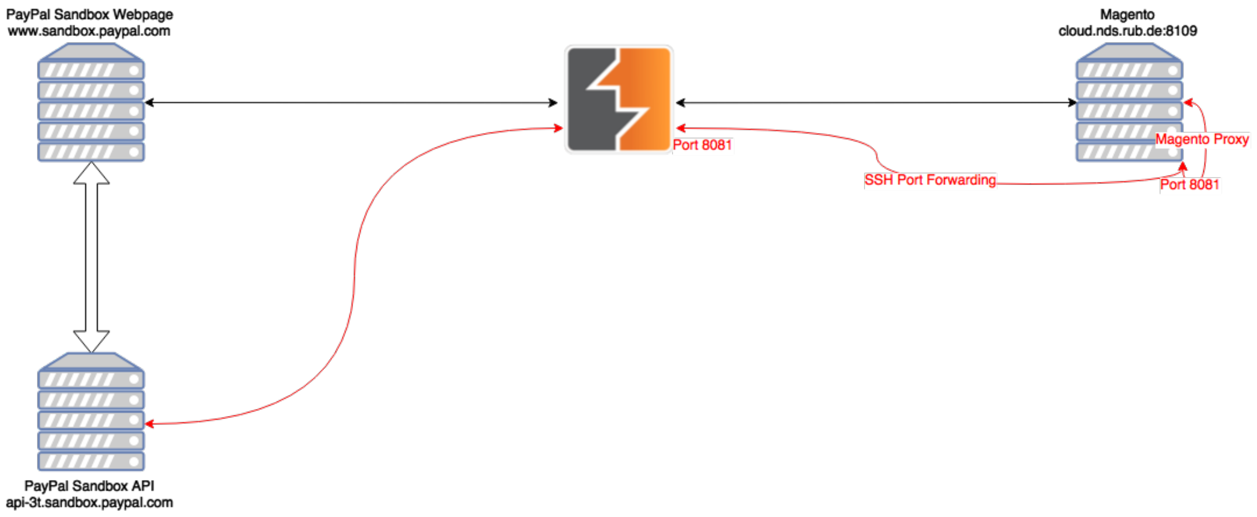


Figure 4.1.: Magento setup with Burp
Burp icon taken from [3]

4.2. Protocol Analysis

PayPals developer documentation contains an explanation of the *Express Checkout* [13]. On this page is also a graphic which explains the message flow between the merchant server, the user and the *PayPal* servers. It is shown in Figure 4.2.

By applying our methodology, we are able to reconstruct the message flow along with the required parameters which can be seen in Figure 4.3.

4.2.1. Explanation of individual messages

The messages are now explained by stepping through one example checkout process.

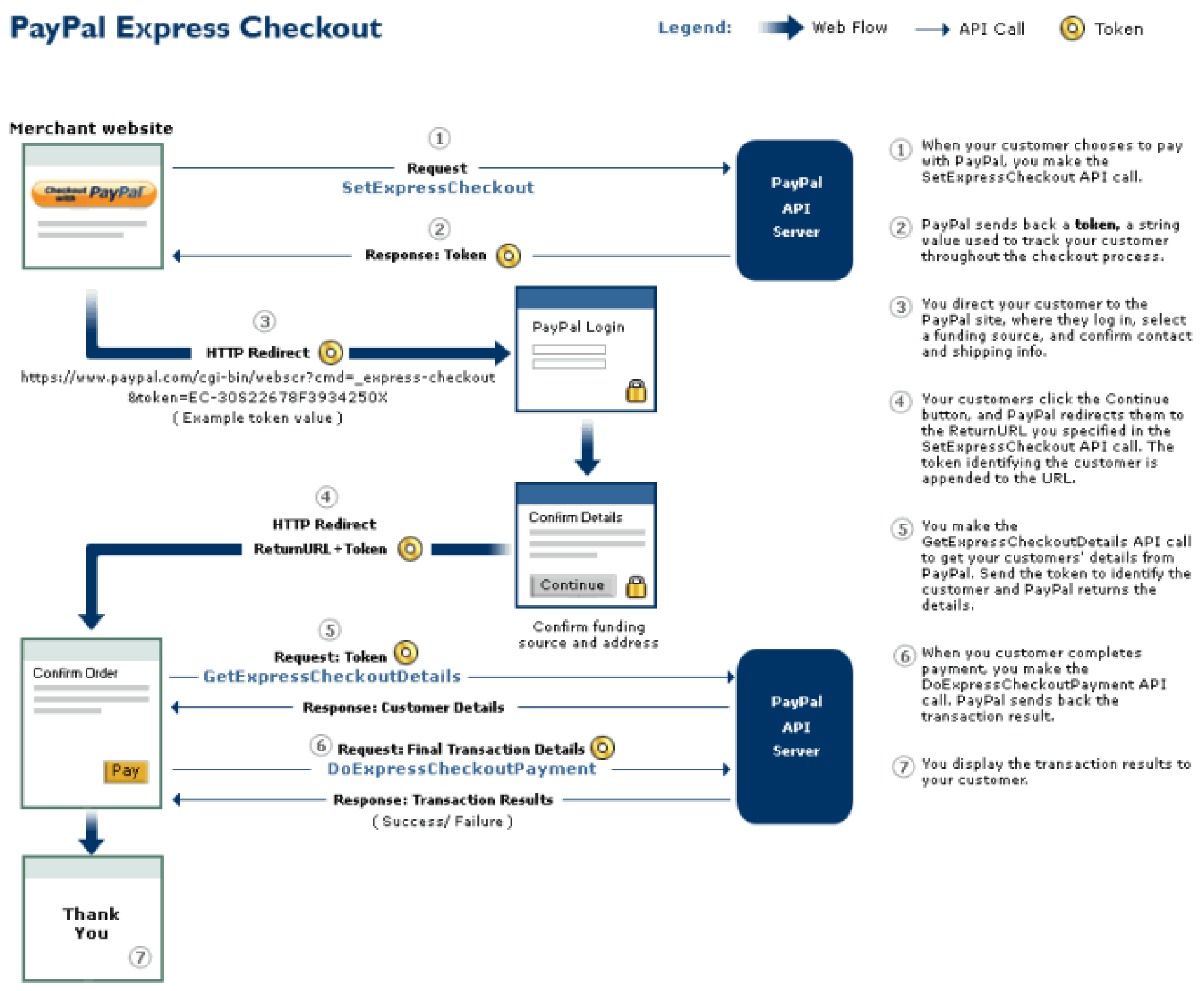
Message 1.a The user wants to checkout his cart and clicks on the *Checkout with PayPal* button.

Message 2.a *Magento* tells the *PayPal NVP API* where to redirect the user after authorizing the transaction via the parameter `RETURNURL` and requests a token for this transaction.

We learn from the `METHOD=SetExpressCheckout` that the *Express Checkout API* is used. The documentation of this specific method can be found under [19]. We explain the meaning of the submitted parameters according to the documentation in Table 4.1 and Table 4.2. Parameters which we suffixed with `*` are deprecated by a new parameter.

Message 2.b The *PayPal NVP API* provides *Magento* with the token.

The explanation of the parameters is again taken from the official documentation found under [19] and [18]. The meaning of the `CORRLEATIONID` is taken from [14]. For the explanation see Table 4.3.


PayPal Express Checkout
Legend: Web Flow API Call Token
Merchant website
Checkout with PayPal
1 Request SetExpressCheckout
2 Response: Token
PayPal API Server
3 HTTP Redirect
https://www.paypal.com/cgi-bin/webscr?cmd=_express-checkout &token=EC-30S22678F3934250X (Example token value)
PayPal Login
4 HTTP Redirect ReturnURL + Token
Confirm Details
Continue
Confirm funding source and address
Confirm Order
Pay
5 Request: Token GetExpressCheckoutDetails
Response: Customer Details
6 Request: Final Transaction Details DoExpressCheckoutPayment
Response: Transaction Results (Success/ Failure)
PayPal API Server
Thank You 7

1. When your customer chooses to pay with PayPal, you make the SetExpressCheckout API call.
2. PayPal sends back a **token,** a string value used to track your customer throughout the checkout process.
3. You direct your customer to the PayPal site, where they log in, select a funding source, and confirm contact and shipping info.
4. Your customers click the Continue button, and PayPal redirects them to the ReturnURL you specified in the SetExpressCheckout API call. The token identifying the customer is appended to the URL.
5. You make the GetExpressCheckoutDetails API call to get your customers' details from PayPal. Send the token to identify the customer and PayPal returns the details.
6. When you customer completes payment, you make the DoExpressCheckoutPayment API call. PayPal sends back the transaction result.
7. You display the transaction results to your customer.

Figure 4.2.: PayPal: Message Flow
picture taken from [13]

`METHOD`	has to be `SetExpressCheckout` for this message
`AMT*`	total cost for transaction
`RETURNURL`	where to redirect the buyer when he decides to pay with PayPal
`CANCELURL`	where to redirect the buyer after refusing to pay with PayPal
`USER, PWD, SIGNATURE`	API values which have been entered in the *Magento* configuration
`VERSION`	release number of the API

Table 4.1.: SetExpressCheckout - required parameters

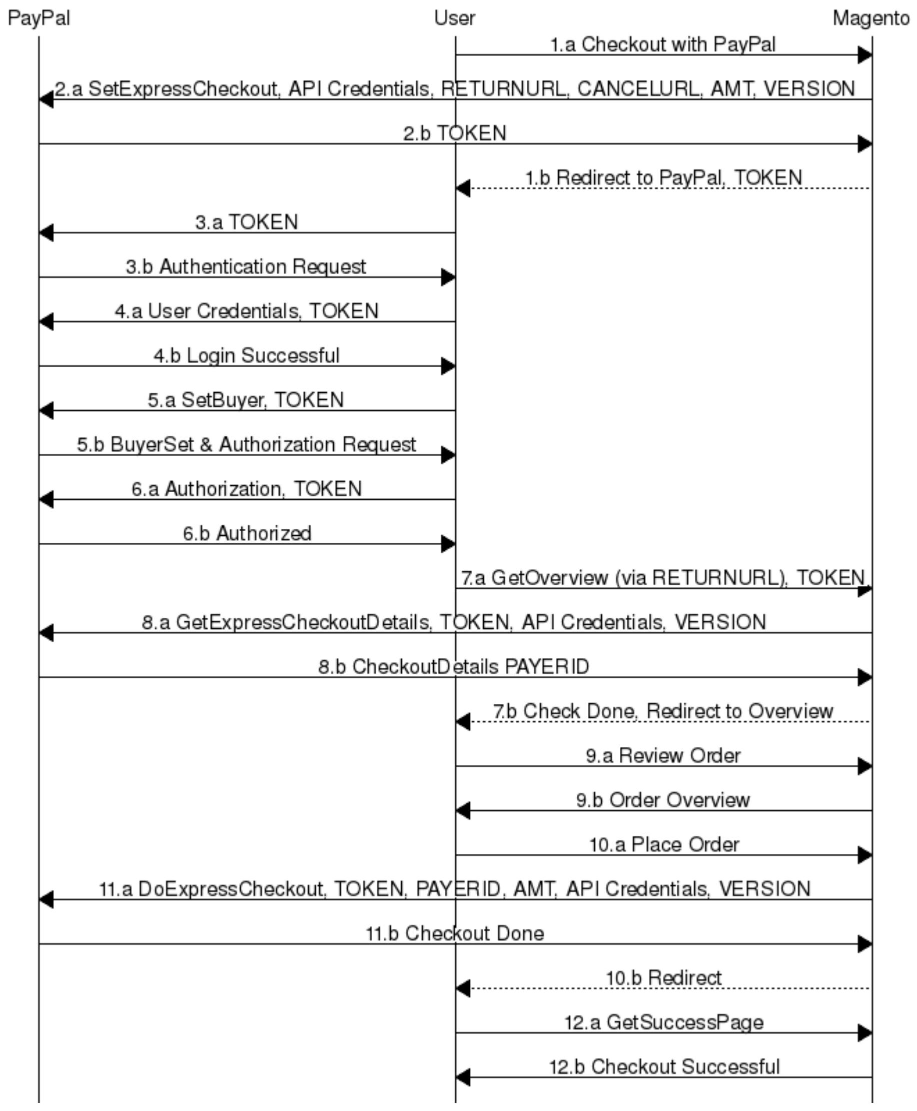

PayPal
User
Magento
1.a Checkout with PayPal
2.a SetExpressCheckout, API Credentials, RETURNURL, CANCELURL, AMT, VERSION
2.b TOKEN
1.b Redirect to PayPal, TOKEN
3.a TOKEN
3.b Authentication Request
4.a User Credentials, TOKEN
4.b Login Successful
5.a SetBuyer, TOKEN
5.b BuyerSet & Authorization Request
6.a Authorization, TOKEN
6.b Authorized
7.a GetOverview (via RETURNURL), TOKEN
8.a GetExpressCheckoutDetails, TOKEN, API Credentials, VERSION
8.b CheckoutDetails PAYERID
7.b Check Done, Redirect to Overview
9.a Review Order
9.b Order Overview
10.a Place Order
11.a DoExpressCheckout, TOKEN, PAYERID, AMT, API Credentials, VERSION
11.b Checkout Done
10.b Redirect
12.a GetSuccessPage
12.b Checkout Successful

Figure 4.3.: PayPal: Reconstructed Message Flow with required parameters

PAYMENTACTION*	can be Sale, Authorization or Order
CURRENCYCODE*	3 character currency code, has to be persistent for all following API calls when set
INVNUM*	invoice or tracking number set by the shop
SOLUTIONTYPE	can be Sole or Mark
GIROPAYSUCCESSURL, GIROPAYCANCELURL	same as RETURNURL and CANCELURL but for giropay
BANKTXNPENDINGURL	RETURNURL for bank transfers
SHIPPINGAMT*	total shipping cost for this order
TAXAMT*	sum of taxes for this order
NOSHIPPING	possible values are 0, 1, 2
ITEMAMT*	sum of item costs for this order
L_AMTn*	Cost of the item number **n**
L_NAMEn*	name of the item **n**
L_QTYn*	quantity of the item **n**
BUTTONSOURCE	identification code for third party apps to identify transactions

Table 4.2.: SetExpressCheckout - optional parameters

TOKEN	identifier for this transaction, is appended to RETURNURL and CANCELURL, expires after 3 hours
TIMESTAMP	date and time in UTC/GMT format, stating when the API operation has been carried out
CORRELATIONID	unique identifier for this API invocation
ACK	indicates success, warnings and errors
VERSION	release number of the API
BUILD	minor of the API

Table 4.3.: SetExpressCheckout - response parameters

Message 1.b *Magento* answers with a redirect response containing the token towards *PayPal*.

Message 3.a The user follows the redirect to *PayPal* and transmits the token.

```
GET /cgi-bin/webscr?cmd=_express-checkout&token=EC-7Y8263384F8650924
Host: www.sandbox.paypal.com
```

Message 3.b *PayPal* displays a login page and sets multiple cookies.

The most interesting one is `x-csrf-jwt`. The `jwt` part tells us that it is a *JSON Web Token (JWT)* [20]. Another JWT is also transmitted as `x-csrf-jwt` header. We decode both JWTs using [8]. The cookie decodes to the following values:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
{
  "token": "
      VKLiZJmxq3ev3x94aN9CDQyuS472c5WS9iqTnQ16G5Ds8zr5MHoH3crMqcbWU8ac8AYEBawUMc6j
       SqwWwts0KikyCVHf1aGKQoi55h24idZ8Mibuv0gfDO2Fzvr2RN4LBNBYMgs-
      scFAuBHF91aclCfjnOpSAfxMCAjltuCp7ILnIjL9I2uZUVGZIlS",
  "iat": 1466945350,
  "exp": 1466948950
}
```

The header token decodes to:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
{
  "token": "SzA6IOT25bbgXDK-
      AofHTtX66b8Dh2KE8SRhYIH17UaI1WUPHvfIYzpM1UtChIkjuCAXqROTtbyuSKnLeS00zTI_CQch
       Ag6yfqnbR-
      cRZimXwb76NokpyZ1CpljwDEjglLPrEb1VkIAL5ftOQdXSifSk7GsCLgq2jLSoF_7gk5GVgq-
      cg8UVwknTkWy",
  "iat": 1466945350,
  "exp": 1466948950
}
```

We note that the `typ`, `alg`, `iat` and `exp` have the same values. Interestingly the tokens have different values. The in-depth analysis of these JWTs is not part of this thesis and all following JWTs will not be noted in the remaining messages.

Message 4.a The user enters his credentials and clicks on the login button.

Both JWTs are transmitted as part of the request. The JSON part of the request contains the email and password of the buyer and the token. The `calc` and `csci` parameters are not explained and neither a search in the *PayPal* documentation nor a internet search yielded results.

Message 4.b *PayPal* tells the user that the login was successful.

PayPal sets a cookie `login_email` containing the email of the buyer. The `nonce` does not appear in any other messages sent. The `buyer_id` is the unique payer identification. The value of `calc` differs from the one in the previous message. For `rlog` and `server` no explanation can be found. Our guess is that `rlog` specifies an unique indicator for this transaction in a logfile on the *PayPal* servers and that `server` denotes which *PayPal* server was used for the transaction.

Message 5.a The browser automatically issues a request to *PayPal*. Without this request, the token can not be authorized in a later step. We assume that this request is used to bind the `token` to the ID of the account.

```
/webapps/hermes/api/batch/setbuyer
Host: www.sandbox.paypal.com
```

In the JSON part of the request, we find the three keywords `eConsent`, `buyereligibility` and `createCheckoutSession`. The latter 2 are bound to this session by the token and are therefore more interesting for further analysis.

Message 5.b *PayPal* shows the details of the transaction and requests its authorization.

For each of the keywords mentioned in Message 5.a, the response has an according answer section. The most interesting one is `createCheckoutSession`. Under the `plan` and `fundingOptions` we see the `amount` keyword.

Message 6.a The user authorizes the transaction.

The interesting part is the URI which contains the token:

```
POST /webapps/hermes/api/checkout/EC-7Y8263384F8650924/session/authorize
Host: www.sandbox.paypal.com
```

Message 6.b *PayPal* confirms the authorization.

Message 7.a The browser issues a request to get an overview of the order with the `token` and `PayerID` appended to the `RETURNURL` *Magento* provided in Message 2.a.

```
GET /paypal/express/return/?token=EC-7Y8263384F8650924&PayerID=VQAGGS4PDGA8A
Host: cloud.nds.rub.de:8109
```

`METHOD`	must be `GetExpressCheckoutDetails`
`TOKEN`	
`VERSION`	release number of the API
`USER, PWD, SIGNATURE`	transmit the API values which have been entered in the configuration
`BUTTONSOURCE`	identification code for third party apps to identify transactions

Table 4.4.: GetExpressCheckoutDetails - parameters

Message 8.a *Magento* issues a request to the *PayPal API* to get the transaction details.

The type of the request is `GetExpressCheckoutDetails` which can be seen by looking at the method type. The according documentation can be found under [15]. For the explanation of the parameters see Table 4.4.

Message 8.b *PayPal* responds with the details.

We explain only values that have not been already explained in Message 2.a and Message 2.b. All parameters which are deprecated by a new parameter are sent along with the new parameter. For the explanation see Table 4.5.

Message 7.b *Magento* answers with a redirect to the review page.

Message 9.a The browser requests the overview page.

Message 9.b *Magento* serves the overview page.

Message 10.a The user clicks on `Place Order`.

Message 11.a *Magento* signals the NVP API to do the transaction.

The final message that the server transmits to the `PayPal NVP API` is of type `DoExpressCheckoutPayment` and is documented in [12].

We only explain values that have not been already explained in Message 2.a, Message 2.b, Message 8.a and Message 8.b. For the explanation of the parameters see Table 4.6, Table 4.7 and Table 4.8.

Message 11.b The *PayPal NVP API* confirms the transaction. For the explanation see Table 4.9.

Message 10.b *Magento* sends a redirect to the success page.

Message 12.a The browser requests the success page.

Message 12.b *Magento* serves the success page.

`BILLINGAGREEMENTACCEPTEDSTATUS`	indicator if the buyer accepted the billing agreement for recurring payments
`CHECKOUTSTATUS`	indicates checkout status, can be any of `PaymentActionNotInitiated`, `PaymentActionFailed`, `PaymentActionInProgress`, `PaymentActionCompleted`
`EMAIL`	buyer email address
`PAYERID`	unique identifier for this account
`PAYERSTATUS`	can be `verified` or `unverified`
`FIRSTNAME`	first name of buyer
`LASTNAME`	last name of buyer
`COUNTRYCODE`	two character indicator of country according to ISO 3166
`CURRENCYCODE, PAYMENTREQUEST_0_CURRENCYCODE`	
`AMT, PAYMENTREQUEST_0_AMT`	
`ITEMAMT, PAYMENTREQUEST_0_ITEMAMT`	
`SHIPPINGAMT, PAYMENTREQUEST_0_SHIPPINGAMT`	
`HANDLINGAMT, PAYMENTREQUEST_0_HANDLINGAMT`	handling costs for this order
`TAXAMT, PAYMENTREQUEST_0_TAXAMT`	
`INVNUM, PAYMENTREQUEST_0_INVNUM`	
`INSURANCEAMT, PAYMENTREQUEST_0_INSURANCEAMT`	insurance costs for shipping
`SHIPDISCAMT, PAYMENTREQUEST_0_SHIPDISCAMT`	shipping discount
`PAYMENTREQUEST_0_SELLERPAYPALACCOUNTID`	seller email address
`INSURANCEOPTIONOFFERED, PAYMENTREQUEST_0_INSURANCEOPTIONOFFERED`	indicator if insurance is available as checkout option, can be `false` or `true`
`ADDRESSSTATUS, PAYMENTREQUEST_0_ADDRESSSTATUS`	can be `none`, `Confirmed` or `Unconfirmed`
`L_NAMEn, L_PAYMENTREQUEST_0_NAME0`	
`L_QTYn, L_PAYMENTREQUEST_0_QTY0`	
`L_TAXAMT0, L_PAYMENTREQUEST_0_TAXAMT0`	
`L_AMT0, L_PAYMENTREQUEST_0_AMT0`	
`PAYMENTREQUESTINFO_0_ERRORCODE`	payment error code

Table 4.5.: GetExpressCheckoutDetails - response

METHOD	has to be DoExpressCheckoutPayment
TOKEN	
PAYERID	
AMT	
VERSION	release number of the API
USER	name of user for the API
PWD	password for the user of the API
SIGNATURE	signature for the user of the API

Table 4.6.: DoExpressCheckoutPayment - required parameters

PAYMENTACTION	
CURRENCYCODE	if it was used in previous requests
L_NAMEn	(required when L_PAYMENTREQUEST_n_ITEMCATEGORYm is passed)
L_QTYn	(required when L_PAYMENTREQUEST_n_ITEMCATEGORYm is passed)
L_AMTn	(required when L_PAYMENTREQUEST_n_ITEMCATEGORYm is passed)

Table 4.7.: DoExpressCheckoutPayment - conditional parameters

BUTTONSOURCE	identification code for third party apps to identify transactions
NOTIFYURL	URL for retrieving Instant Payment Notifications (IPN)
RETURNFMFDETAILS	flag to indicate if you want to see the results of the Fraud Management Filters (FMF)
SHIPPINGAMT	
TAXAMT	
EMAIL	buyer email address
FIRSTNAME	first name of buyer, Not documented
LASTNAME	last name of buyer, Not documented
COUNTRYCODE	country of buyer, Not documented
STREET	Not documented
ADDROVERRIDE	Not documented
ITEMAMT	

Table 4.8.: DoExpressCheckoutPayment - optional parameters

`TOKEN`	
`SUCCESSPAGEREDIRECTREQUESTED`	redirect the buyer to sign up for PayPal after a successful transaction, can be `false` or `true`
`TRANSACTIONID, PAYMENTINFO_0_TRANSACTIONID`	unique ID for this transaction
`TRANSACTIONTYPE, PAYMENTINFO_0_TRANSACTIONTYPE`	can be `cart` or `express-checkout`
`PAYMENTTYPE, PAYMENTINFO_0_PAYMENTTYPE`	can be `none`, `echeck` or `instant`
`ORDERTIME, PAYMENTINFO_0_ORDERTIME`	timestamp of the payment
`AMT, PAYMENTINFO_0_AMT`	
`FEEAMT, PAYMENTINFO_0_FEEAMT`	PayPal fee deducted for this transaction
`TAXAMT, PAYMENTINFO_0_TAXAMT`	
`CURRENCYCODE, PAYMENTINFO_0_CURRENCYCODE`	
`PAYMENTSTATUS, PAYMENTINFO_0_PAYMENTSTATUS`	status of the payment
`PENDINGREASON, PAYMENTINFO_0_PENDINGREASON`	reason the transaction is pending
`REASONCODE, PAYMENTINFO_0_REASONCODE`	reason for a reversal
`PROTECTIONELIGIBILITY, PAYMENTINFO_0_PROTECTIONELIGIBILITY`	indicates if the merchant is protected by Pay-Pals merchant protection
`PAYMENTINFO_0_PROTECTIONELIGIBILITYTYPE`	kind of protection
`INSURANCEOPTIONSELECTED`	indicates if the buyer chose insurance, can be `true` or `false`
`SHIPPINGOPTIONISDEFAULT`	indicates if the buyer chose the default option, can be `true` or `false`
`PAYMENTINFO_n_SECUREMERCHANTACCOUNTID`	unique PayPal ID of the merchant
`PAYMENTINFO_n_ERRORCODE`	payment error code
`PAYMENTINFO_n_ACK`	specific error message

Table 4.9.: DoExpressCheckoutPayment - response

4.3. Security Analysis

After analyzing the general protocol structure, we wish to analyze the security of the protocol.

The `TOKEN` is the parameter which holds the complete payment information and identifies a finished payment at the shop. This makes it a very interesting target for attacks.
We develop several tests to identify in which ways the token can be attacked.

4.3.1. Test: Token Validity

In this test we determine if the `token` expires after the transaction has been authorized but the token was not consumed. The idea is that a malicious shop could bruteforce tokens that are authorized but have not been claimed by another shop. This can be the case if there is an error within the order handling at the shop or if the return to the shop via Message 7.a is interrupted. According to the documentation the token should only be valid for 3 hours [19]. We are successful if the token is valid for a significantly longer time.

We follow the protocol until Message 11.a and intercept this message. We note the time and after 3 hours and 5 minutes to compensate for timing differences we forward the message. We receive the following response:

```
...ACK=Success&...&PAYMENTINFO_0_ACK=Success...
```

The `ACK=Success` part shows that the transaction has been completed successfully. We confirm this by logging into our sandbox account on `sandbox.paypal.com` and viewing the transaction history and notice that the transaction was completed.

Repeating the test after intercepting the message for 4 hours shows that the token is no longer accepted. This means that the token expires roughly 3-4 hours after it has been issued.

4.3.2. Test: Unauthorized Token

For the next test we try if unauthorized tokens for which the buyer is already set, can be used in the transaction. After the user logs in with his credentials, the buyer is automatically set for this token. If we are able to complete the transaction without an authorized token, the protocol would be fatally flawed because as soon as a user could be tricked to login into his paypal account it could be completed. This means that a malicious shop does not have to wait for the explicit authorization simply by having a user logging in to PayPal. For this test we complete one checkout completely so that we have a valid example of Message 11.a. Then we perform a new checkout until Message 5.b and replace the old token with the token from the new checkout in the Message 11.a we wish to repeat. If the response contains `ACK=SUCCESS` we have successfully bypassed the authorization.

PayPal responds with an error message stating that the token can not be used because it is not authorized.

4.3.3. Test: Token Swapping

Most attacks from [23] use the idea that the transaction token can be freely exchanged by another one. Following these ideas we conduct several experiments in which we swap the currently used token by another. It could for example be possible to put expensive items into a cart but override the token with another one which was issued for cheap items in the cart, depending on the application logic the shop could flag the expensive cart as paid because it received a valid token. This would obviously incur a financial loss for the shop. For these types of token swapping attacks we assume the attacker to be a malicious user. We consider these attacks successful if it is possible to complete the order with a different token than the one which was issued by PayPal for this exact transaction.

Inserting an old token, with which a transaction has already been completed, into the redirection location of Message 1.b leads to *PayPal* showing a page which states that the transaction has already been completed.

Inserting an old token into Message 7.a in order to bypass the check at PayPal makes *Magento* throw an error which states that a wrong PayPal Express Checkout Token was specified.

For the next experiment we complete a checkout process until Message 9.b. Then we edit the cart and change the amount of the item from 1 to 2. Next we click on the `Checkout with PayPal` button which starts the protocol flow a second time. In Message 3.a from the new protocol flow we insert the authorized token from the previous run. *Magento* responds with the same error message as before.

Next, we try to swap the token sent in Message 1.b by another fresh token from another protocol run. We can advance the protocol until Message 7.a where *Magento* issues the same error, stating that a wrong PayPal Express Checkout Token was specified.

From these results we induce that once *Magento* receives the token in Message 2.b, it binds the token to the user session internally and expects to receive this exact token. This effectively thwarts all attacks that require the tokens to be exchangeable.

4.3.4. Test: PayerID Manipulation

For this test we manipulate the `PayerID` that is sent along with the `token` in Message 11.a. For this test we use the malicious shop attacker model in combination with the malicious user attacker model. By changing the `PayerID` to the `PayerID` of another account we hope to charge the customer with the second `PayerID` instead of e.g. our own. If this is possible the attacker can repeatedly authorize `tokens` for his account but change the `PayerID` to that of another customer in Message 11.a. This has as consequence that the shop can effectively steal money from accounts that never

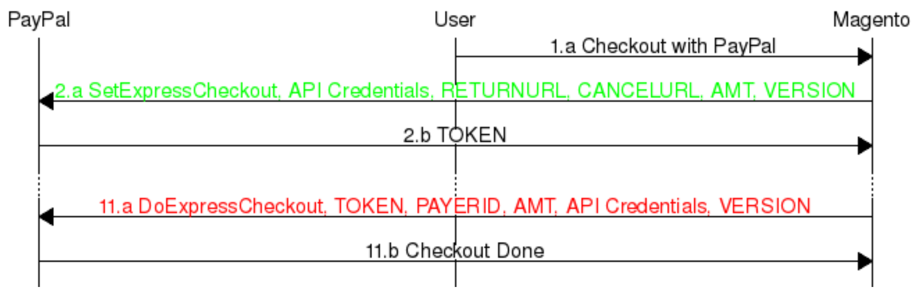

PayPal
User
Magento
1.a Checkout with PayPal
2.a SetExpressCheckout, API Credentials, RETURNURL, CANCELURL, AMT, VERSION
2.b TOKEN
11.a DoExpressCheckout, TOKEN, PAYERID, AMT, API Credentials, VERSION
11.b Checkout Done

Figure 4.4.: PayPal: Authorization attack

participated in the transaction. We determine if this is successful by inspecting the answer of the `PayPal NVP API`.

Fortunately PayPal recognizes the mismatch between the account who authorized the `token` and the account which is given as source of the payment. This can be seen in the following response:

```
TIMESTAMP=2016%2d09%2d25T14%3a58%3a39Z&CORRELATIONID=e7b89cc4a6c49&ACK=Failure&
    VERSION=72%2e0&BUILD=25395022&L_ERRORCODE0=10421&L_SHORTMESSAGE0=This%20
    Express%20Checkout%20session%20belongs%20to%20a%20different%20customer%2e&
    L_LONGMESSAGE0=This%20Express%20Checkout%20session%20belongs%20to%20a%20different%20customer
    %2e%20%20Token%20value%20mismatch%2e&L_SEVERITYCODE0=Error
```

4.3.5. Attack: Authorization not bound to amount

The idea is to check if the authorization the user issues in Message 6.a is bound to the explicit amount or if the amount itself is not part of the authorization. If the latter is the case then it is possible for a malicious shop to charge more than the user was willing to pay. After a transaction the user gets an confirmation email for the order in which the effective transmitted amount is displayed. Considering that not all users check these confirmation emails for the correct amount, the attack can go undetected for a long time. An attacker could also choose to charge only slightly higher prices, e.g. 0.50 - 1€ in order to conceal the attack even better. For this test we complete the checkout process until Message 11.a. We intercept this message and change the value of the `AMT` parameter to 10 because this is the parameter that holds the information about the total amount when the shop finally executes the transaction. Then we submit the modified message and inspect the response in Message 11.b. If it contains `ACK=SUCCESS` and a manual check in the transaction history of the used sandbox accounts shows the increased amount, the attack is successful.

The attack is successful because we get a response from PayPal which indicates success and the manual verification in the account also shows that the increased amount was deduced from the balance.

We visualize the attack in Figure 4.4.

Original request:

```
TOKEN=EC-1RL31436KF794245A&PAYERID=VQAGGS4PDGA8A&PAYMENTACTION=Sale&AMT=1.00&
    CURRENCYCODE=EUR&BUTTONSOURCE=Magento_Cart_Community&NOTIFYURL=https%3A%2F%2
    Fcloud.nds.rub.de%3A8109%2Fpaypal%2Fipn%2F&RETURNFMFDETAILS=1&SHIPPINGAMT
    =0.00&ITEMAMT=1.00&TAXAMT=0.00&L_NAME0=AES&L_QTY0=1&L_AMT0=1.00&EMAIL=
    ssoanonym2-magento-buyer%40gmail.com&FIRSTNAME=magento&LASTNAME=buyer&
    COUNTRYCODE=DE&STREET=&ADDROVERRIDE=1&METHOD=DoExpressCheckoutPayment&VERSION
    =72.0&USER=ssoanonym2-magento-merchant_api1.gmail.com&PWD=6HYBHXX97RZRVAZF&
    SIGNATURE=AFcWxV21C7fd0v3bYYYRCpSSRl31AuhJAdthBWjCln7nvDtcTqvVH-sR
```

Modified request:

```
TOKEN=EC-1RL31436KF794245A&PAYERID=VQAGGS4PDGA8A&PAYMENTACTION=Sale&AMT=10.00&
    CURRENCYCODE=EUR&BUTTONSOURCE=Magento_Cart_Community&NOTIFYURL=https%3A%2F%2
    Fcloud.nds.rub.de%3A8109%2Fpaypal%2Fipn%2F&RETURNFMFDETAILS=1&SHIPPINGAMT
    =0.00&ITEMAMT=1.00&TAXAMT=0.00&L_NAME0=AES&L_QTY0=1&L_AMT0=1.00&EMAIL=
    ssoanonym2-magento-buyer%40gmail.com&FIRSTNAME=magento&LASTNAME=buyer&
    COUNTRYCODE=DE&STREET=&ADDROVERRIDE=1&METHOD=DoExpressCheckoutPayment&VERSION
    =72.0&USER=ssoanonym2-magento-merchant_api1.gmail.com&PWD=6HYBHXX97RZRVAZF&
    SIGNATURE=AFcWxV21C7fd0v3bYYYRCpSSRl31AuhJAdthBWjCln7nvDtcTqvVH-sR
```

Countermeasure The best countermeasure is to save the AMT the user authorized along with the rest of the information on the PayPal servers and check the stored AMT against the AMT that is sent in Message 11.a. If they do not match the transaction should be aborted because this indicates that the user authorized an AMT that differs from the one the shop is trying to charge.

We notified *PayPal* about this vulnerability got the response that security issues in the sandbox are not eligible for a bugbounty. Nevertheless they seem to have implemented a fix because after testing the attack once again, we got the following response:

```
TOKEN=EC%2d1RL31436KF794245A&SUCCESSPAGEREDIRECTREQUESTED=false&TIMESTAMP=2016%2
    d09%2d21T10%3a54%3a56Z&CORRELATIONID=3fe1bd92a1648&ACK=Failure&VERSION=72%2e0&
    BUILD=000000&L_ERRORCODE0=10413&L_SHORTMESSAGE0=Transaction%20refused%20
    because%20of%20an%20invalid%20argument%2e%20See%20additional%20error%20
    messages%20for%20details%2e&
    L_LONGMESSAGE0=The%20totals%20of%20the%20cart%20item%20amounts%20do%20not%20match%20order
    %20amounts
```

In particular L_LONGMESSAGE0 states that the total of the cart does not match the total of the order. This still leaves the question if the production API is vulnerable to the attack. We suspect that even if it was vulnerable they will have fixed it by now.

5. *Woocommerce's PayPal Payments Standard* Integration

5.1. Testing Environment

The testing environment is the same as for *Magento*, consisting of the same virtual machine, this time with *Woocommerce 2.6.1* installed. We set a global proxy for outgoing requests by adding the following lines to `wp-config.php` which can be found in the root directory of *Wordpress*:

```
define('WP_PROXY_HOST', '127.0.0.1');
define('WP_PROXY_PORT', '8082');
```

Again we have to disable the SSL verification so that we can man-in-the-middle the connection between *PayPal* and *Woocommerce*. To this end we modify the file `/var/www/wp-includes/class-http.php` and change line 197 from `true` to `false`:

```
'sslverify' => false,
```

We forward the port 8082 by issuing the following command:

```
ssh -N -R 8082:localhost:8082 <IP of Virtual Machine>
```

We are not able to intercept Message 6.a because the request is coming from *PayPal* directly. To proxy it nevertheless we use a clever trick.

The trick requires us to add the following line to `/etc/ssh/sshd_config` because SSH does not forward remote incoming connections per default:

```
GatewayPorts yes
```

We restart `sshd` to enable this option.

For the trick itself we first create a Burp proxy listener on port 8083. Next we modify the port of the `notifyurl` in Message 1.b to 40109 which gets forwarded to our VM on port 40000. Finally we forward port 40000 from the VM to our local *Burp* instance by issuing the following command:

```
ssh -N -R 40000:localhost:8083 <IP of Virtual Machine>
```

After intercepting and tampering with the request we have to forward it to the original destination at the VM, otherwise *Woocommerce* will not continue the checkout process.

Figure 5.1 illustrates the finished setup.

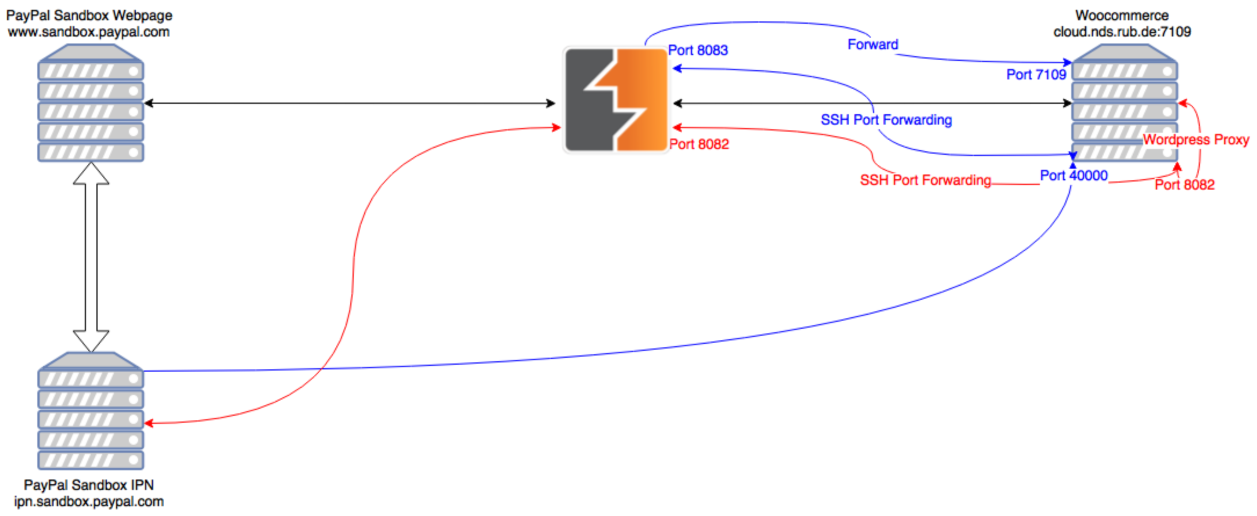


Figure 5.1.: Woocommerce setup with Burp
Burp icon taken from [3]

5.2. Protocol Analysis

After noticing in our initial tests that *Woocommerce* does not have any information that the payment has been completed because the checkout process finished even if we do not visit the return-page of the shop, we concluded that there has to be a mechanism that notifies *Woocommerce* about the successful payment. In the settings of the *PayPal* checkout integration of *Woocommerce* we found the following message beneath the `Debug Log` option:

```
Log PayPal events, such as IPN requests, inside /var/www/wordpress/wp-content/
    uploads/wc-logs/paypal-8e798d4985e2806d18016f2d3abaaffa.log
```

By entering `PayPal IPN` into our favourite search machine, we quickly found an explanation [16]. There it also states that 4 messages are exchanged to deliver the payment notification. We integrate these 4 messages as messages Message 6.a, Message 6.b, Message 7.a and Message 6.a into our diagram.

By applying our methodology, we are able to reconstruct the message flow along with the required parameters which can be seen in Figure 5.2.

5.2.1. Explanation of individual messages

The messages are now explained by stepping through one example checkout process.

Message 1.a The user clicks the checkout button after having filled in the billing details which consist of `First Name`, `Last Name`, `Email Address`, `Phone`, `Country`, `Address`, `Postcode` and `Town`, which are then transmitted to *Woocommerce*.

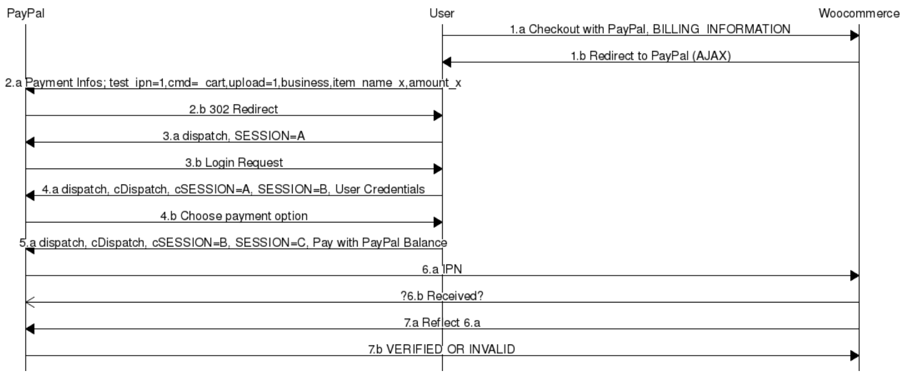

PayPal
User
Woocommerce
1.a Checkout with PayPal, BILLING INFORMATION
1.b Redirect to PayPal (AJAX)
2.a Payment Infos; test ipn=1,cmd= cart,upload=1,business,item name x,amount_x
2.b 302 Redirect
3.a dispatch, SESSION=A
3.b Login Request
4.a dispatch, cDispatch, cSESSION=A, SESSION=B, User Credentials
4.b Choose payment option
5.a dispatch, cDispatch, cSESSION=B, SESSION=C, Pay with PayPal Balance
6.a IPN
?6.b Received?
7.a Reflect 6.a
7.b VERIFIED OR INVALID

Figure 5.2.: PayPal: Reconstructed Message Flow with required Parameters

`test_ipn=1`	indicates that this message is used for testing and sent to the sandbox
`cmd=_cart`	shopping cart
`upload`	tells PayPal, that a shopping cart from a third party is uploaded
`business`	PayPalID or associated email address of the merchant
`item_name_x`	item name for item x, use *1* to specify name for the whole cart
`amount_x`	amount for item x, use *1* to specify amount for the whole cart
`no_note`	whether the buyer can send a note to the seller or not

Table 5.1.: Redirect to PayPal - required parameters

Message 1.b The server responds with a message which triggers an AJAX redirect to *PayPal*. The redirect URL contains all the information needed to complete the checkout, the parameters are explained in the next message.

Message 2.a The user follows the redirect and sends a request to *PayPal*. After searching for `no_note` on `developer.paypal.com` we found [7] which explains the meaning of all variables. The explanation of the individual variables is taken from there and additionally from [16]. For the explanation see Table 5.1 and Table 5.2.

Message 2.b *PayPal* responds with a redirect. The parameters of the redirectURL are explained in the next message.

Message 3.a The user follows the redirect. We are not able to find a documentation for `cmd=_flow`, `SESSION` or `dispatch` in the PayPal documentation.
We assume that the redirect is made so that *PayPal* can bind the transaction to a unique identifier. The occurrence of `dispatch` with the exact same value in later messages makes it plausible that this is the parameter the transaction is bound to. Tampering with either one leads to an error and checkout can not be completed.

Message 3.b *PayPal* delivers the login page.

`quantity_x`	quantity of item number x, not needed when used for complete cart
`currency_code`	currency used for the payment, defaults to USD
`charset`	set character encoding for the billing information and PayPal Login page
`return`	where to redirect the user after a successful transaction
`rm`	sets the return method for the `return` parameter, defaults to 0
`cancel_return`	where to redirect the user when the checkout is canceled
`page_style`	page style for checkout pages
`paymentaction`	indicate if the payment is for a final sale or a authorization for a final sale
`bn`	identifier for the source of the button click
`invoice`	pass-through variable to identify this order for the shop application
`custom`	variable which is passed back to the merchant via the IPN message
`notify_url`	target URL for IPN messages
`first_name, last_name, company, address1, address2, city, state, zip, country, email, night_phone_b, day_phone_b`	used to fill out the buyer information automatically
`no_shipping`	whether to prompt for a shipping address or not
`item_number`	pass-through variable to track the order, passed back on payment completion

Table 5.2.: Redirect to PayPal - optional parameters

Message 4.a The user logs in with his credentials.
`dispatch` and `currentDispatch` contain the previous `dispatch` variable, which is also transmitted via GET parameter. The value of the `SESSION` variable is transmitted as `currentSession`.

Message 4.b *PayPal* presents the user with a choice of payment options.

Message 5.a The user chooses to use his *PayPal Balance* for this transaction. `dispatch` can be found as part of the POST URL, `currentDispatch` and `dispatch` parameter as part of the body.

After this message, no more user interaction is required for the checkout to complete, thus we leave out the rest of the messages which are submitted to or from the user.

Message 6.a *PayPal* sends a *Instant Payment Notification (IPN)* to *Woocommerce*. The IPN message for this example checkout contains the parameters which are explained in Table 5.3. They are explained according to [17].

Message 6.b According to [16], *Woocommerce* answers with an empty response to advance the IPN protocol. We are not able to confirm this behavior as we neither saw this message in Burp nor found evidence for this message in the source code of *Woocommerce*.

Message 7.a *Woocommerce* sets `cmd=_notify_validate` as the first parameter of the message, adds the contents of Message 6.a and sends it back to *PayPal* for verification, as evidenced by [16].
The only parameters that are **not** checked are the following:

- `ipn_track_id`
- `mc_handling1`
- `mc_shipping1`
- `tax1`
- `mc_gross_1`
- `test_ipn`

This means that all the essential parameters we want to tamper with, e.g. quantities and prices, are protected.

Message 7.b According to [16], *PayPal* checks if the contents are identical with the contents it sent in Message 6.a and responds with `VERIFIED` if this is the case. Otherwise the response is `INVALID`.

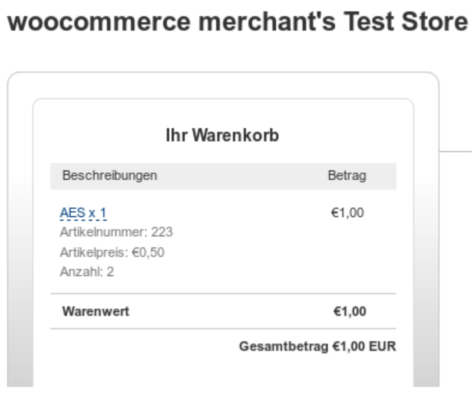
Woocommerce does some checks on the IPN parameters against the information stored in the order. Additionally to our methodology we found these conditions by checking the source code of:

mc_gross	total amount of the payment before substraction of the transaction fee
invoice	pass-through variable to identify this order for the shop application
protection_eligibility	seller protection type
item_numberx	Pass-through variable to track the purchase
payer_id	unique customer ID
tax	amount of tax
payment_date	timestamp of the payment
payment_status	status of the payment
charset	character set
mc_shipping	total shipping amount
mc_handling	total handling amount
first_name	first name of the customer
mc_fee	transaction fee
notify_version	version number
custom	variable which is passed back to the merchant via the IPN message
payer_status	whether the payer has verified his account
business	PayPalID or associated email address
num_cart	number of items in the *PayPal* shopping cart
mc_handlingx	handling amount of item number x
verify_sign	Encrypted string used to validate the authenticity of the transaction
payer_email	email address of the customer
mc_shippingx	shipping amount of item number x
taxx	tax amount of item number x
txn_id	transaction identifier for this transaction
payment_type	type of the payment
last_name	last name of the customer
item_namex	name of item number x
receiver_email	primary email address of the recipient
payment_fee	only used for USD
quantityx	amount of items number x
receiver_id	unique ID of the recipient
txn_type	kind of transaction for which the message was sent
mc_gross_x	amount of the payment for item number x
mc_currency	currency of the payment
residence_country	ISO 3166 country code
test_ipn	whether this is a test message or not
transaction_subject	
payment_gross	only used for USD
ipn_track_id	only for internal use for *PayPal*

Table 5.3.: PayPal IPN message - parameters

```
/var/www/wordpress/wp-content/plugins/woocommerce/includes/gateways/paypal/
    includes/class-wc-gateway-paypal-ipn-handler.php
```

In particular the following conditions are checked:
The response from PayPal has to be `VERIFIED`.
`txn_type` has to be conform with the one specified in the order, in our case this is `cart`.
`mc_currency` has to be the same as in the order.
`mc_gross` which denotes the overall amount of the transaction has to be the same as specified in the order.
`receiver_email` has to be the email of the merchant.
If any of these checks fail *Woocommerce* automatically places the order on hold.


woocommerce merchant's Test Store

Ihr Warenkorb

Beschreibungen	Betrag
AES x 1 Artikelnummer: 223 Artikelpreis: €0,50 Anzahl: 2	€1,00
Warenwert	€1,00

Gesamtbetrag €1,00 EUR

Figure 5.3.: PayPal: tampered amount and quantity

5.3. Security Analysis

We have already seen that both PayPal and Woocommerce perform various checks on the parameters. This means that we cannot tamper with messages Message 6.a, Message 6.b, Message 7.a or Message 7.b. This limits the scope of our testing to Message 2.a.
We cannot tamper with the parameters `cmd`, `currency_code` and `business` because they are compared against the values `txn_type` `mc_currency`, and `receiver_email` respectively, which are stored by Woocommerce on a per order basis. The parameter `mc_gross` is also compared with the result from the IPN message. We assume that the gross of the total order is calculated by PayPal by computing the sum of all `quantity_x` times `amount_x` for all items `x`. This assumption leads us to the first test.

5.3.1. Test: Tampering with quantity and amount

The idea is to change the `quantity` and `amount` of the items in Message 2.a in way that the overall gross stays the same. If the checkout process completes successfully we can use the discrepancy between which items are shown to the user and which items are bought for further attacks. For this test we do not need a specific attacker model. We follow the normal checkout process until Message 2.a and intercept this message. We change the value of the following parameters from

```
amount_1=1.00
quantity_1=1
```

to:

```
amount_1=0.50
quantity_1=2
```

The corresponding login page at PayPal is shown in Figure 5.3.

We complete the checkout process and notice that Woocommerce automatically advances the order. The results of this test can be used in the following attack.

5.3.2. Attack: Foreign Payment for your Order

The idea of this attack is to let another user pay for your order. For this attack we use the CSRF-attacker that wants to checkout his cart at the shop and we can trick the user into authorizing the transaction because the cart is something that he would buy for himself. The attacker follows the checkout process until Message 2.a and intercepts this message which transmits the information that is needed for the checkout. He then uses the target of the GET request to construct a link that points to this exact location. If a user clicks on the link the attacker provided, he will be redirected to PayPal and see the cart that the attacker wishes to checkout. A sample attacker cart can be seen in Figure 5.4a. This cart is not very convincing since it shows the items that the attacker wants to buy. Luckily the `item_name_1` is not checked by Woocommerce and we can freely set the name of the item. In fact we could also add additional items, as long as the individual prices and amounts are adapted so that the overall gross stays the same.
Assume that the attacker loves cryptography and wants to check out one item of type `AES`. Further assume that we have an user who likes chocolate. The attacker can now tamper with section 5.2.1 so that instead of one item of type `AES` for a price of 1€, the cart will show one item of type `Chocolate 1` for a price of 0.50€ and one item of type `Chocolate 2` for a price of another 0.50€. In the crafting process the attacker changes the following parameters:

```
item_name_1=Chocolate+1
quantity_1=1
amount_1=0.50
```

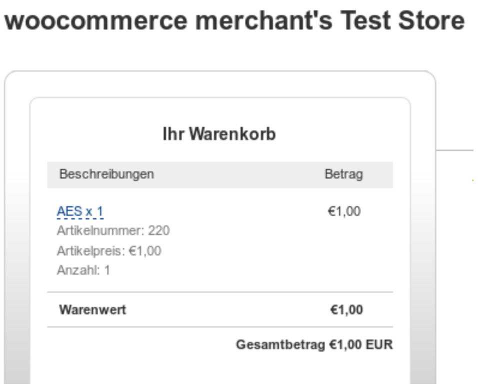
Furthermore he adds the following new parameters:

```
item_name_2=Chocolate+2
quantity_2=1
amount_2=0.50
```

To not confuse the user, the attacker also strips out the parameter `item_number_1`.
As a result the user is presented with the cart that is shown in Figure 5.4b.

The checkout completes successfully because Woocommerce does not check individual items, amounts and prices but only compares the total amounts of the orders.

Countermeasure An obvious countermeasure for Woocommerce is to check if the items, quantities and amounts of the order are the same that PayPal transmits in Message 6.a. Should this not be the case, Woocommerce has to hold the automatic processing of the order and notify PayPal about the discrepancy of the order. PayPal in turn should revert the transaction immediately. This countermeasure does not defend against against tricking the user into paying for the exact same cart that the attacker wants.



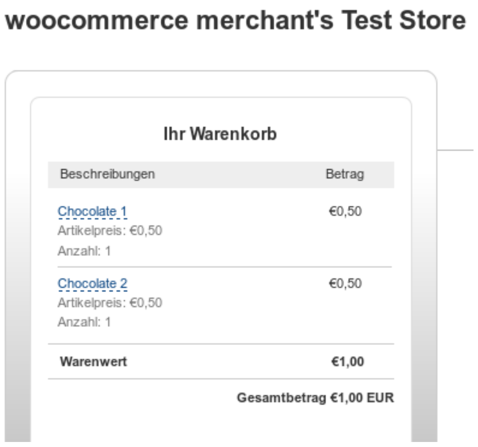
woocommerce merchant's Test Store

Ihr Warenkorb

Beschreibungen	Betrag
AES x 1 Artikelnummer: 220 Artikelpreis: €1,00 Anzahl: 1	€1,00
Warenwert	**€1,00**

Gesamtbetrag €1,00 EUR

(a) PayPal: attacker cart



woocommerce merchant's Test Store

Ihr Warenkorb

Beschreibungen	Betrag
Chocolate 1 Artikelpreis: €0,50 Anzahl: 1	€0,50
Chocolate 2 Artikelpreis: €0,50 Anzahl: 1	€0,50
Warenwert	**€1,00**

Gesamtbetrag €1,00 EUR

(b) PayPal: masqueraded attacker cart

Figure 5.4.: PayPal cart representation

6. Conclusion

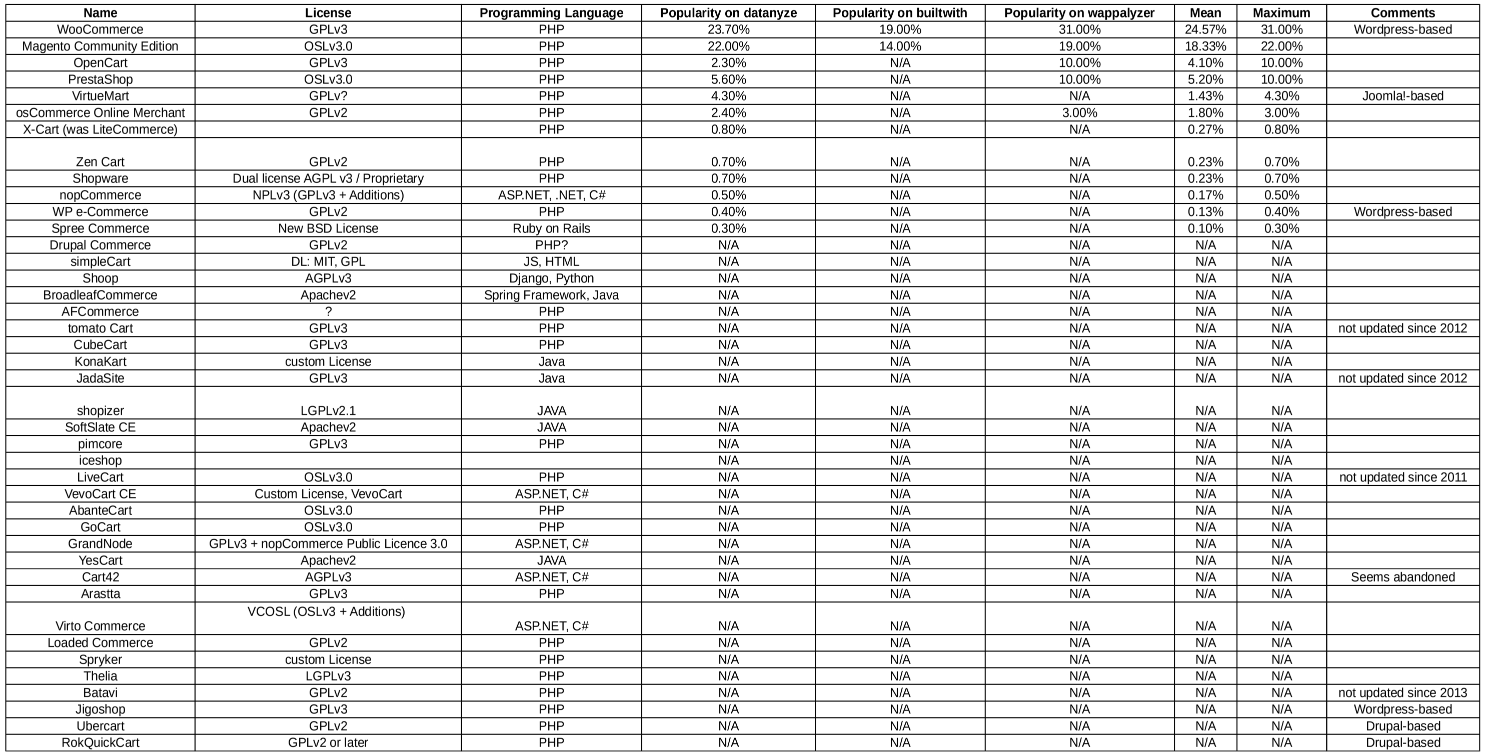
In this thesis we analyzed the integration of one Cashier-as-Service solution in two different self-hostable webshops. After collecting information about various webshop solutions, we analyzed Magentos integration of PayPals Express Checkout and Woocommerces integration of PayPal Payments Standard. Before the analysis of the underlying protocols we had to recover the protocols. For this purpose we developed a methodology for the recovery of an unknown protocol. After having successfully recovered the respective protocols we analyzed them.
The protocols have in common that the critical information about the payment in not transmitted via the user but via a direct channel between PayPal and the shop. This is an improvement from the mechanisms of the past in which the user could often tamper with the payment information as seen in [23].
While Magento and Woocommerce have checks in place to prevent common attacks, we found that PayPal has a vulnerability in their payment authorization when using the attacker model *malicious shop*. The vulnerability made it possible for an malicious shop to deduct an arbitrary amount from the PayPal account of the victim. We notified PayPal about this issue and got as answer that vulnerabilities in the sandbox are not eligible for a bugbounty. Although in the mean time they seem to have implemented a fix.
In the case of Woocommerce we found an attack that lets an attacker change the appearance of the shopping cart at PayPal. By shaping the appearance to another cart that a victim is likely to buy, the attacker can let another user pay for his shopping via phishing. The only limitation is that the total amount of the transaction has to be the same.
These vulnerabilities show that even among the most popular CaaS and webshops there are still issues regarding a safe implementation. From these results we expect that lesser known shops and cashiers have even more critical flaws.

Future Work

Please note that the analysis was conducted against *PayPals Sandbox API* which can differ from the production API. This means that the malicious shop attack must also be validated against the real API.
Further research can be conducted by looking into the integration of Magento and Woocommerce or other webshops with different Cashier-as-a-Service providers, e.g. Amazon. Another topic are shop-hosting solutions like *shopify* [22]. Also PayPals authentication and authorization process has to be researched in detail.

A. Appendix

A.1. Spreadsheet Excerpt



Name	License	Programming Language	Popularity on datanyze	Popularity on builtwith	Popularity on wappalyzer	Mean	Maximum	Comments
WooCommerce	GPLv3	PHP	23.70%	19.00%	31.00%	24.57%	31.00%	Wordpress-based
Magento Community Edition	OSLv3.0	PHP	22.00%	14.00%	19.00%	18.33%	22.00%	
OpenCart	GPLv3	PHP	2.30%	N/A	10.00%	4.10%	10.00%	
PrestaShop	OSLv3.0	PHP	5.60%	N/A	10.00%	5.20%	10.00%	
VirtueMart	GPLv?	PHP	4.30%	N/A	N/A	1.43%	4.30%	Joomla!-based
osCommerce Online Merchant	GPLv2	PHP	2.40%	N/A	3.00%	1.80%	3.00%	
X-Cart (was LiteCommerce)		PHP	0.80%	N/A	N/A	0.27%	0.80%	
Zen Cart	GPLv2	PHP	0.70%	N/A	N/A	0.23%	0.70%	
Shopware	Dual license AGPL v3 / Proprietary	PHP	0.70%	N/A	N/A	0.23%	0.70%	
nopCommerce	NPLv3 (GPLv3 + Additions)	ASP.NET, .NET, C#	0.50%	N/A	N/A	0.17%	0.50%	
WP e-Commerce	GPLv2	PHP	0.40%	N/A	N/A	0.13%	0.40%	Wordpress-based
Spree Commerce	New BSD License	Ruby on Rails	0.30%	N/A	N/A	0.10%	0.30%	
Drupal Commerce	GPLv2	PHP?	N/A	N/A	N/A	N/A	N/A	
simpleCart	DL: MIT, GPL	JS, HTML	N/A	N/A	N/A	N/A	N/A	
Shoop	AGPLv3	Django, Python	N/A	N/A	N/A	N/A	N/A	
BroadleafCommerce	Apachev2	Spring Framework, Java	N/A	N/A	N/A	N/A	N/A	
AFCommerce	?	PHP	N/A	N/A	N/A	N/A	N/A	
tomato Cart	GPLv3	PHP	N/A	N/A	N/A	N/A	N/A	not updated since 2012
CubeCart	GPLv3	PHP	N/A	N/A	N/A	N/A	N/A	
KonaKart	custom License	Java	N/A	N/A	N/A	N/A	N/A	
JadaSite	GPLv3	Java	N/A	N/A	N/A	N/A	N/A	not updated since 2012
shopizer	LGPLv2.1	JAVA	N/A	N/A	N/A	N/A	N/A	
SoftSlate CE	Apachev2	JAVA	N/A	N/A	N/A	N/A	N/A	
pimcore	GPLv3	PHP	N/A	N/A	N/A	N/A	N/A	
iceshop			N/A	N/A	N/A	N/A	N/A	
LiveCart	OSLv3.0	PHP	N/A	N/A	N/A	N/A	N/A	not updated since 2011
VevoCart CE	Custom License, VevoCart	ASP.NET, C#	N/A	N/A	N/A	N/A	N/A	
AbanteCart	OSLv3.0	PHP	N/A	N/A	N/A	N/A	N/A	
GoCart	OSLv3.0	PHP	N/A	N/A	N/A	N/A	N/A	
GrandNode	GPLv3 + nopCommerce Public Licence 3.0	ASP.NET, C#	N/A	N/A	N/A	N/A	N/A	
YesCart	Apachev2	JAVA	N/A	N/A	N/A	N/A	N/A	
Cart42	AGPLv3	ASP.NET, C#	N/A	N/A	N/A	N/A	N/A	Seems abandoned
Arastta	GPLv3	PHP	N/A	N/A	N/A	N/A	N/A	
Virto Commerce	VCOSL (OSLv3 + Additions)	ASP.NET, C#	N/A	N/A	N/A	N/A	N/A	
Loaded Commerce	GPLv2	PHP	N/A	N/A	N/A	N/A	N/A	
Spryker	custom License	PHP	N/A	N/A	N/A	N/A	N/A	
Thelia	LGPLv3	PHP	N/A	N/A	N/A	N/A	N/A	
Batavi	GPLv2	PHP	N/A	N/A	N/A	N/A	N/A	not updated since 2013
Jigoshop	GPLv3	PHP	N/A	N/A	N/A	N/A	N/A	Wordpress-based
Ubercart	GPLv2	PHP	N/A	N/A	N/A	N/A	N/A	Drupal-based
RokQuickCart	GPLv2 or later	PHP	N/A	N/A	N/A	N/A	N/A	Drupal-based

Figure A.1.: Spreadsheet Excerpt - Open source solutions ordered by maximum popularity

Bibliography

[1] D. Akhawe et al. “Towards a Formal Foundation of Web Security”. In: *Proc. 23rd IEEE Computer Security Foundations Symp.* July 2010, pp. 290–304. DOI: 10.1109/CSF.2010.27.

[2] *BuiltWith Technology Lookup.* URL: https://trends.builtwith.com/shop (visited on 2016-05-08).

[3] *Burp Icon.* URL: http://forum.portswigger.net/thread/1188/icon-make-burp-pretty-dock (visited on 2016-09-22).

[4] *Burp Suite.* URL: https://portswigger.net/burp/.

[5] *Cart2Cart - Automated Shopping Cart Migration Service.* URL: http://www.shopping-cart-migration.com/supported-carts (visited on 2016-05-08).

[6] *Datanyze.* URL: http://www.datanyze.com/market-share/e-commerce-platforms/Alexa%20top%201M (visited on 2016-05-08).

[7] *HTML Variables for PayPal Payments Standard.* URL: https://developer.paypal.com/docs/classic/button-manager/integration-guide/ButtonManagerHTMLVariables/ (visited on 2016-08-02).

[8] *jwt.io JSON Web Token Decoder.* URL: https://jwt.io/ (visited on 2016-07-17).

[9] Christian Mainka, Vladislav Mladenov, et al. “Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE. 2016, pp. 321–336.

[10] *OpenID.* URL: https://openid.net/ (visited on 2016-07-16).

[11] *paydirekt.* URL: https://www.paydirekt.de/kaeufer/index.html (visited on 2016-09-10).

[12] *PayPal: DoExpressCheckoutPayment API.* URL: https://developer.paypal.com/docs/classic/api/merchant/DoExpressCheckoutPayment_API_Operation_NVP/ (visited on 2016-06-26).

[13] *PayPal: Express Checkout Overview.* URL: https://developer.paypal.com/docs/classic/express-checkout/overview-ec/ (visited on 2016-06-13).

[14] *PayPal FAQ: CorrelationID.* URL: https://www.paypal-knowledge.com/infocenter/index?page=content&widgetview=true&id=FAQ1087 (visited on 2016-06-26).

[15] *PayPal: GetExpressCheckoutDetails API.* URL: https://developer.paypal.com/docs/classic/api/merchant/GetExpressCheckoutDetails_API_Operation_NVP/ (visited on 2016-06-25).

[16] *PayPal: Instant Payment Notification (IPN).* URL: https://developer.paypal.com/docs/classic/ipn/integration-guide/IPNIntro/ (visited on 2016-07-30).

[17] *PayPal: NVP and PDT Variables.* URL: https://developer.paypal.com/docs/classic/ipn/integration-guide/IPNandPDTVariables/ (visited on 2016-08-08).

[18] *PayPal: NVP API.* URL: `https://developer.paypal.com/webapps/developer/docs/classic/api/NVPAPIOverview/` (visited on 2016-06-25).

[19] *PayPal: SetExpressCheckout API.* URL: `https://developer.paypal.com/docs/classic/api/merchant/SetExpressCheckout_API_Operation_NVP/` (visited on 2016-06-25).

[20] *RFC 7519 - JSON Web Token (JWT).* URL: `https://tools.ietf.org/html/rfc7519` (visited on 2016-07-17).

[21] *SAML.* URL: `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security` (visited on 2016-07-17).

[22] *Shopify.* URL: `https://www.shopify.com/`.

[23] Rui Wang et al. "How to Shop for Free Online–Security Analysis of Cashier-as-a-Service Based Web Stores". In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 465–480.

[24] *Wappalyzer.* URL: `https://wappalyzer.com/categories/ecommerce` (visited on 2016-05-08).

[25] Stefan Weinfurtner et al. "Erfolgsfaktor Payment–Der Einfluss der Zahlungsverfahren auf Ihren Umsatz". In: *Aktuelle Ergebnisse zum Bezahlverhalten der Endkunden aus dem Projekt E-Commerce-Leitfaden, 2nd ed., ibi research, Regensburg* (2013).