IOActive 1426 Elliot Avenue W, Seattle, WA 98119
(866) 760.0222 I info@ioactive.com

August 27, 2021

Pando Inc.
12/F, San Toi Building
137-139 Connaught Road
Central West, Hong Kong, 999077, HK

Code Review and Penetration Test

Pando engaged IOActive to perform a code review and penetration test of its service.

Three IOActive consultants conducted the assessment from the 7th of June to the 2nd of July 2021 for a total effort of eight resource weeks. The consultants focused on the most common web application security vulnerabilities as listed in the OWASP Top Ten.

The following branches were in scope for this assessment:

- https://github.com/fox-one/compound/tree/audit
- https://github.com/fox-one/pando/tree/audit

Results: IOActive identified one high-risk vulnerability in the service.

Pando addressed the issue and engaged IOActive to perform remediation validation testing. One IOActive security consultant conducted a retest on the 9th of August 2021 and confirmed the reported issue was properly fixed.

Respectfully,

DocuSigned by:
Jennifer Steffens
E0FA5D9E7A2D46C...

Jennifer Steffens
Chief Executive Officer
IOActive, Inc.

DS
IOA LCA Approved - KM

Technical Summary

#MP-07 - Compound - Borrow Repayment Transaction May Fail After Modifying Borrow Balance [FIXED]

Host(s) / File(s)	Compound worker/snapshot/borrow_repay.go
Category	Authentication
Testing Method	Code Review
Tools Used	Visual Studio Code
Likelihood	Medium (3)
Impact	Critical (5)
Total Risk Rating	High (15)
Remediation Status	Fixed
Effort to Fix	Low

Threat and Impact

IOActive found that the borrow repayment logic in `Payee.handleRepayEvent()` suffered from a logic bug that could result in a 'borrow balance' being updated as 'repaid' or 'partially repaid' in the event of edge case failure conditions. As such, the error exists because logic within this function updates the balance on the relevant `borrow` object before actually generating and storing the transactions necessary for the repayment to take place. If error conditions occur during the system's attempts to generate and store these transactions, the result will be that the borrow would appear to have been repaid, yet no repayment transactions will end up taking place.

The discussion below briefly outlines the relevant code path.

The `payee` worker in Compound is responsible for dispatching handling for a number of events, including `ActionTypeRepay`. This action is used to repay a `borrow` (i.e. a loan). When an item of this type is encountered by the worker, Compound calls into `Payee.handleRepayEvent()`, which lives in the `worker/snapshot/borrow_repay.go` file.

The code fragments below, from `Payee.handleRepayEvent()`, show the relevant borrow balance being updated, with failure conditions actually being possible after these updates are made. Note the annotations of the form “`[n] potential failure point.`”

```
//update borrow info
      borrowBalance, e := w.borrowService.BorrowBalance(ctx,
borrow, market)
      if e != nil {
            log.Errorln(e)
            return e
        }
        realRepaidBalance := repayAmount
        redundantAmount :=
repayAmount.Sub(borrowBalance).Truncate(8)
        newBalance := borrowBalance.Sub(repayAmount)
        newIndex := market.BorrowIndex
        if newBalance.LessThanOrEqual(decimal.Zero) {
            newBalance = decimal.Zero
            newIndex = decimal.Zero
            realRepaidBalance = borrowBalance
        }

        if output.ID > borrow.Version {
            borrow.Principal = newBalance.Truncate(16)
            borrow.InterestIndex = newIndex.Truncate(16)
            if e = w.borrowStore.Update(ctx, borrow,
output.ID); e != nil {     // [1] update the 'borrow' with
the new balance that is correct after the user-specified
amount is deducted due to repayment
                log.Errorln(e)
                return e
            }
        }

....

        if output.ID > market.Version {
            market.TotalBorrows =
market.TotalBorrows.Sub(realRepaidBalance).Truncate(16)
            market.TotalCash =
market.TotalCash.Add(realRepaidBalance).Truncate(16)
            if e = w.marketStore.Update(ctx, market,
output.ID); e != nil {    // [2] potential failure point
                log.Errorln(e)
                return e
            }
        }

        // market transaction
        marketTransaction :=
core.BuildMarketUpdateTransaction(ctx, market,
foxuuid.Modify(output.TraceID, "update_market"))
        if e = w.transactionStore.Create(ctx,
marketTransaction); e != nil {     // [3] potential failure
point
            log.WithError(e).Errorln("create transaction
error")
            return e
        }

        // add transaction
        extra := core.NewTransactionExtra()
        extra.Put(core.TransactionKeyBorrow, core.ExtraBorrow{
            UserID:        borrow.UserID,
            AssetID:       borrow.AssetID,
            Principal:     borrow.Principal,
                InterestIndex: borrow.InterestIndex,
        })
        transaction := core.BuildTransactionFromOutput(ctx,
userID, followID, core.ActionTypeRepay, output, extra)
        if e = w.transactionStore.Create(ctx, transaction); e
!= nil {        // [4] potential failure point
                log.WithError(e).Errorln("create transaction
error")
                return e
        }
```

The operation at [1] updates the 'borrow balance' as per the amount the user is attempting to repay (e.g. complete or partial balance), and this change is committed to the 'borrow store'. However, failure points exist at [2], [3], and [4]; if failure does happen at one of these points, transactions to actually repay the 'borrow' may not actually be created in the system, hence the 'borrow' will appear to have been repaid, yet repayment will not actually happen.

These operations at these failure points revolve around database commit operations, with the relevant 'stores' being SQLite databases managed by the GORM library. Whilst failures at these points may be unlikely under normal operating conditions, simple database commit operations may occasionally fail under high load or when a system experiences edge-case conditions; for example, low memory conditions may result in memory allocation failures, which in turn could result in a database commit operation failing overall.

Recommendations

Commit changes to the borrow store only after generating and storing the relevant repayment transactions.

Retest Results

2021-08-09: Fixed

The code now makes sure that the database is not updated if the payment fails.

```
// handle borrow repay event
func (w *Payee) handleRepayEvent(ctx context.Context, output
*core.Output, userID, followID string, body []byte) error {

    log := logger.FromContext(ctx).WithField("worker",
"borrow_repay")

    repayAmount := output.Amount
    assetID := output.AssetID

    log.Infoln(":asset:", output.AssetID, "amount:",
repayAmount)

    market, e := w.marketStore.Find(ctx, assetID)
    if e != nil {
            return e
    }

    if market.ID == 0 {
            return w.handleRefundEvent(ctx, output, userID,
followID, core.ActionTypeRepay, core.ErrMarketNotFound)
```

```
	}

	//update interest
	if e = w.marketService.AccrueInterest(ctx, market,
output.CreatedAt); e != nil {
		log.Errorln(e)
		return e
	}

	borrow, e := w.borrowStore.Find(ctx, userID, market.AssetID)
	if e != nil {
		return e
	}

	if borrow.ID == 0 {
		return w.handleRefundEvent(ctx, output, userID,
followID, core.ActionTypeRepay, core.ErrBorrowNotFound)
	}

	transaction, e := w.transactionStore.FindByTraceID(ctx,
output.TraceID)
	if e != nil {
		return e
	}

	if transaction.ID == 0 {
		if w.marketService.IsMarketClosed(ctx, market) {
			return w.handleRefundEvent(ctx, output, userID,
followID, core.ActionTypeRepay, core.ErrMarketClosed)
		}

		//update borrow info
		borrowBalance, e := w.borrowService.BorrowBalance(ctx,
borrow, market)
		if e != nil {
			log.Errorln(e)
			return e
		}

		newBalance := borrowBalance.Sub(repayAmount)
		newIndex := market.BorrowIndex
		if !newBalance.IsPositive() {
			newBalance = decimal.Zero
			newIndex = decimal.Zero
			repayAmount = borrowBalance
		}

		extra := core.NewTransactionExtra()
		extra.Put("repay_amount", repayAmount.Truncate(16))
		extra.Put("new_balance", newBalance.Truncate(16))
		extra.Put("new_index", newIndex.Truncate(16))
		extra.Put(core.TransactionKeyBorrow, core.ExtraBorrow{
			UserID:        borrow.UserID,
			AssetID:       borrow.AssetID,
			Principal:     newBalance,
			InterestIndex: newIndex,
		})
```

```
        transaction = core.BuildTransactionFromOutput(ctx, 
userID, followID, core.ActionTypeRepay, output, extra)
        if e := w.transactionStore.Create(ctx, transaction); e 
!= nil {
            return e
        }
    }

    var extra struct {
        RepayAmount decimal.Decimal `json:"repay_amount"`
        NewBalance  decimal.Decimal `json:"new_balance"`
        NewIndex    decimal.Decimal `json:"new_index"`
    }

    if e := transaction.UnmarshalExtraData(&extra); e != nil {
        return e
    }

    if output.ID > borrow.Version {
        borrow.Principal = extra.NewBalance
        borrow.InterestIndex = extra.NewIndex
        if e = w.borrowStore.Update(ctx, borrow, output.ID); e 
!= nil {
            log.Errorln(e)
            return e
        }
    }

    if refundAmount := output.Amount.Sub(extra.RepayAmount); 
refundAmount.GreaterThan(decimal.Zero) {
        transferAction := core.TransferAction{
            Source:   core.ActionTypeRepayRefundTransfer,
            FollowID: followID,
        }

        if e := w.transferOut(ctx, userID, followID, 
output.TraceID, assetID, refundAmount, &transferAction); e != 
nil {
            return e
        }
    }

    if output.ID > market.Version {
        market.TotalBorrows = 
market.TotalBorrows.Sub(extra.NewBalance).Truncate(16)
        market.TotalCash = 
market.TotalCash.Add(extra.NewBalance).Truncate(16)
        if e = w.marketStore.Update(ctx, market, output.ID); e 
!= nil {
            log.Errorln(e)
            return e
        }
    }

    return nil
}
```

IOActive Qualifications

IOActive has more than 20 years of experience providing information security consulting services. Established in 1998, IOActive is an industry leader that specializes in:

- IT infrastructure vulnerability assessments and pen tests
- Application security source code and architecture reviews
- ICS/SCADA and smart grid assessments and pen tests
- Emerging market assessments and pen tests (cloud, embedded, automotive, and more)
- Security development lifecycle training and review

IOActive works with many Global 500 companies including organizations in the power and utility, industrial, game, hardware, embedded, retail, financial, media, travel, aerospace, healthcare, high-tech, social networking, cloud, and software development industries.

We provide unequalled technical services, strive to become trusted advisors to our clients, and help them achieve their business and security objectives. We go well beyond off-the-shelf code scanning tools to perform gap analysis on information security policies and protocols. We also conduct deep analyses of information systems, software architecture, and source code using leading information risk and security management frameworks and focused threat models.

Your opponents do not use unsophisticated commercial pen-test tools to undermine your enterprise security. They use the smartest code breakers money can buy to footprint and damage your organization’s brand using advanced, often unknown methods. IOActive's industry experience helps our clients consistently stay ahead of tomorrow's threats.

IOActive attracts people who contribute to the growing body of security knowledge by speaking at elite conferences such as RSA, SANS, SOURCE, Black Hat, InfoSecurity Europe, DEF CON, Blue Hat, and CanSecWest. We also have key advisors like Steve Wozniak and David Lacey, luminaries who affect how security and technology shape our world.